# The Distributed Computing Column

by

**Panagiota Fatourou**

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr

# Lower Bounds and Impossibility Results for Transactional Memory Computing

Hagit Attiya
Department of Computer Science, Techion
Israel
hagit@cs.technion.ac.il

**Abstract**

We overview some impossibility results and lower bounds on the complexity of implementing software transactional memory, and explain their underlying assumptions.

## 1 Introduction

As anyone with a laptop or an Internet connection knows, the multi-core revolution is here, since almost any computing appliance contains several processing cores. With the improved hardware comes the need to harness the power of concurrency, since the processing power of individual cores does not increase. Applications must be restructured in order to reap the benefits of multiple processing units, without paying a hefty price for coordination among them.

It has been argued that writing concurrent applications is significantly more challenging than writing sequential ones, and *Transactional memory* (TM) has been suggested as a way to deal with this difficulty. In the simplest form of TM, the programmer need only wrap code with operations denoting the beginning and end of a transaction. The transactional memory will take care of synchronizing the shared memory accesses so that each transaction seems to execute sequentially and in isolation.

Originally suggested as a hardware platform by Herlihy and Moss [29], TM has resurfaced as a software mechanism a couple of years later. The first software implementation of transactional memory was suggested by Shavit and Touitou [43]; it provided, in essence, support for multi-word synchronization operations on a static set of data items, in terms of a unary operation (LL/SC), somewhat optimized over prior implementations, e.g., [9, 46]. Shavit and Touitou coined the term *software transactional memory* (STM) to describe their implementation.

Only when the termination condition was relaxed to *obstruction freedom* (see Section 2.2), the first STM handling a dynamic set of data items was presented by Herlihy et al. [28]. Work by Rajwar et al., e.g., [38, 41], helped to popularize the TM approach in the programming languages and hardware communities.

Despite its simplicity, or perhaps because of it, transactional memory implementations incur significant cost, as has been discovered in recent theoretical work. This short survey describes several of these impossibility results and lower bounds, and their interaction with various properties of transactional memory.

# 2 Formalizing TM

This section outlines how transactional memory can be formally captured, as well as properties expected of it. A comprehensive in-depth treatment is provided by Guerraoui and Kapałka [25].

The model encompasses at least two levels of abstraction: The high level has *transactions*, each of which is a sequence of *operations* accessing data items. At the low level, the operations are translated into executions in which a sequence of events apply *primitive operations* (or *primitives*) to base objects, containing the data and the meta-data needed for the implementation. (See Figure 1.)

A *transaction* is a sequence of operations executed by a single process on a set of *data items*, shared with other transactions. Data items are accessed by *read* and *write* operations; some systems also support other operations. The interface also includes *try-commit* (*tryC*) and *try-abort* (*tryA*) operations, in which a transaction requests to commit or abort, respectively. If the response of *try-commit* is *commit*, the writes of the transaction are ensured to take effect, and we say that the transaction is *committed*. Any of these operations, not just *try-abort*, may cause the transaction to abort, in which case, none of its writes take effect and we say that the transaction is *aborted*. If the transaction is aborted not in response to *try-abort*, we say that it is *forcibly* aborted.

The collection of data items accessed by a transaction is its *data set*; the items written by the transaction are its *write set*, with the other items being its *read set*.

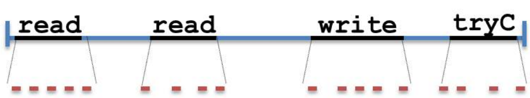


Figure 1: Levels of abstraction in transactional memory.

A *software implementation of transactional memory* (abbreviated *STM*) provides data representation for transactions and data items using *base objects*, and algorithms, specified as primitives on the base objects. These procedures are followed by *asynchronous* processes in order to execute the operations of transactions. The primitives can be simple reads and writes, but also more sophisticated ones, like CAS or DCAS, typically applied to memory locations, which are the base objects for the implementation.

When processes invoke these procedures, in an interleaved manner, we obtain *executions*, in the standard sense of asynchronous distributed computing (cf. [8]). Executions consist of *configurations*, describing a complete state of the system at some point in time, and *events*, describing a single step by an individual process, including an application of a single primitive to base objects (possibly several objects, e.g., in case of DCAS).

The *interval of a transaction* $T$ is the execution interval that starts at the first event of $T$ and ends at the last event of $T$. If $T$ does not have a last event in the execution, then the interval of $T$ is the (possibly infinite) execution interval starting at the first event of $T$. Two transactions *overlap* if their intervals overlap.

## 2.1 Safety: Consistency Properties of TM

An STM is *serializable* if committed transactions appear to execute sequentially, one after the other [39]. An STM is *strictly serializable* if this serialization order preserves the order of non-overlapping transactions [39]. This notion is called *order-preserving serializability* in [47], and is the analogue of *linearizability* [31] for transactions.[1]

*Opacity*, suggested by Guerraoui and Kapałka [23], further demands that even partially executed transactions, which may later abort, must be serializable (in an order-preserving manner). Opacity also accommodates operations beyond read and write.

While opacity is a stronger condition than serializability, *snapshot isolation* [10] is a consistency condition weaker than serializability. Roughly stated, snapshot isolation ensures that all read operations in a transaction return the most recent value as of the time the transaction starts; the write sets of concurrent transactions must be disjoint. (Cf. [47, Definition 10.3].) Riegel et al. [42] proposed to use snapshot isolation for TM.

*Virtual World Consistency* (VWC), defined by Imbs et al. [32], is a weakening of opacity, tailored for transactional memory. VWC allows aborted (and ongoing)

[1] *Linearizability*, like *sequential consistency* [37], talks about implementing abstract data structures, and hence they involve one abstraction—from the high-level operations of the data structure to the low level primitives. It also provides the semantics of the operations, and their expected results at the high-level, on the data structure.

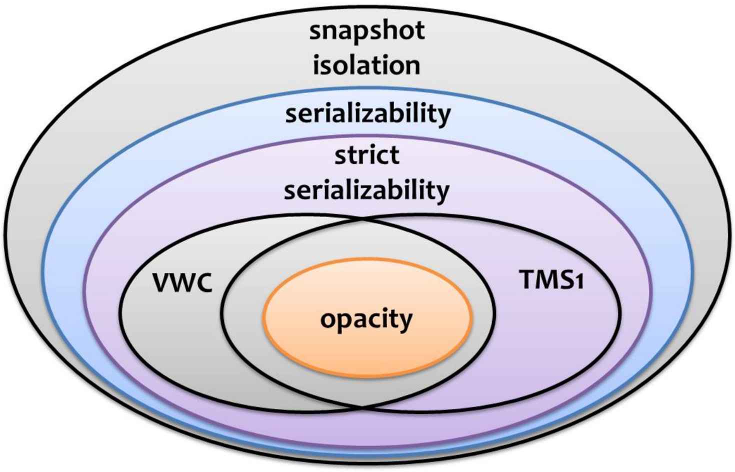


Figure 2: The relations between several TM and database consistency conditions.

transactions to observe *mutually inconsistent* views of the execution, as long as each of them is consistent with some sequential execution of the committed transactions in their "causal past". A related condition, called *Transactional Memory Specification* (referred to as TMS1), was suggested by Doherty et al. [16], also considers each aborted transaction in isolation.

Figure 2 summarizes these conditions and the relations between them. Additional discussion of the relations between various TM and database consistency conditions is given by Attiya and Hans [26].

## 2.2 Progress: Termination Guarantees for TM

One of the innovations of TM is in allowing transactions not to commit, when they are faced with conflicting transactions, namely, transactions that access the same data items. This, however, admits trivial implementations where no progress is ever made. Finding the right balance between nontriviality and efficiency has lead to several progress properties. They are first and foremost distinguished by whether locking is accommodated or not.

When locks are not allowed, the strongest requirement—rarely provided—is of *wait-freedom*, namely, that each transaction has to eventually commit. A weaker property ensures that some transaction eventually commits, or that a transaction commits solo, for long enough time. The last property is called *obstruction-freedom* [28] (see further discussion in [4]).

A *lock-based* STM (e.g., TL2 [15]) is often required to be *(weakly) progressive* [24], namely, a transaction that does not encounter a conflicting transaction must commit. (There is a *conflict* between two transactions, if both of them access the same data item.)

Several lower bounds assume a minimal progress property, ensuring that a transaction terminates successfully if it runs alone, from a situation in which no other transaction is pending. This property is implied both by obstruction freedom and by weak progressiveness.

Related definitions [18, 24, 34] further attempt to capture the distinction between aborts that are necessary in order to maintain the safety properties (e.g., opacity) and *spurious* aborts that are not mandated by the consistency property, and to measure their ratio.

*Strong progressiveness* [24] ensures that even when there are conflicts, some transaction commits. More specifically, an STM is *strongly progressive* if a transaction without *nontrivial* conflicts, namely, a conflict involving at least one write. is not forcibly aborted, and if a set of transactions have nontrivial conflicts on a single item then not all of them are forcibly aborted. (Recall that a transaction is forcibly aborted, when the abort was not requested by a *try-abort* operation of the transaction, i.e., the abort is in response to *try-commit*, *read* or *write* operations.)

*Permissiveness* tries to capture the number of unjustified, *spurious* aborts; it requires a transaction to commit unless doing so violates correctness [20]; said otherwise, this means that a transaction can abort or block only if committing may violate correctness. A weaker condition, given by Fan et al. [40], says that an STM is *multi-version (MV)-permissive* if a transaction is forcibly aborted (not because it requests to abort) only if it is an update transaction that has a nontrivial conflict with another update transaction.

Strong progressiveness and MV-permissiveness are incomparable: The former allows a read-only transaction to abort, if it has a conflict with another update transaction, while the latter does not guarantee that at least one transaction is not forcibly aborted in case of a conflict.

Figure 3 shows the relations between these progress conditions.

**Remark 1.** *Strictly speaking, these properties are not liveness properties in the traditional sense [36], since they can be checked in finite executions.*

## 2.3 Performance Indicators

There has been some theoretical attempts to predict how well will TM implementations scale, resulting in definitions that postulate behaviors that are expected to yield superior performance.

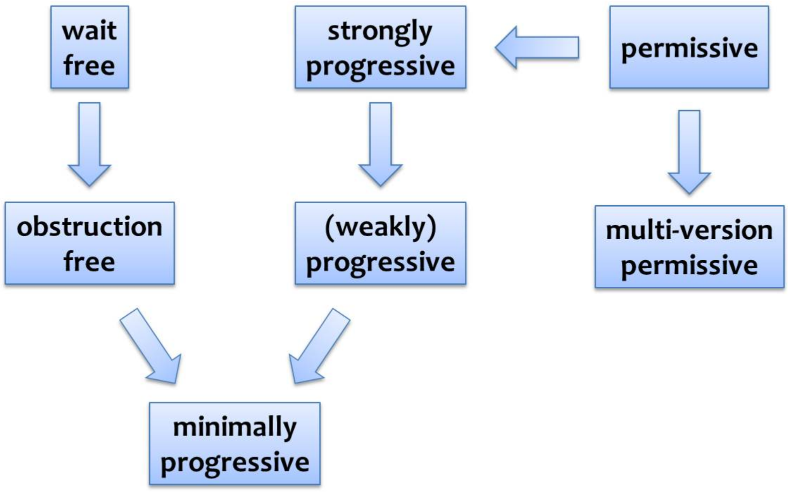


Figure 3: Relations between progress conditions for transactional memory.

#### 2.3.1 Disjoint-Access Parallelism

The most accepted such notion is *disjoint-access parallelism*, capturing the requirement that unrelated transactions progress *independently*, even if they occur at the same time. That is, an implementation should not cause two transactions, which are unrelated at the high-level, to simultaneously access the same low-level shared memory.

We explain what it means for two transactions to be *unrelated* through a conflict graph that represents the relations between transactions. The *conflict graph* of an execution interval $I$ is an undirected graph, where vertices represent transactions whose execution intervals intersect, and edges connect transactions that share a data item. Two transactions $T_1$ and $T_2$ are *disjoint access* if there is no path between the vertices representing them in the conflict graph of their execution intervals; they are *strictly disjoint access* if there is no edge between these vertices.

Below is the conflict graph for six transactions: T1 with data set $\{A, B, C\}$, T2 with data set $\{A, D\}$, T3 with data set $\{D, E\}$, T4 with data set $\{F, L\}$, T5 with data set $\{L\}$ and T6 with data set $\{J\}$.

In this example, the data sets of T1 and T2 intersect, as do the data sets of T2 and T3, while the data sets of T1 and T3 do not intersect. Hence, T1 and T3 are strictly disjoint access, but they are not disjoint access.

Two events *contend* on a base object $o$ if they both access $o$, and at least one of them applies a nontrivial primitive to $o$. (A primitive is *nontrivial* if it may change the value of the object, e.g., a write or CAS; otherwise, it is *trivial*, e.g., a read.) Transactions *concurrently contend* on a base object $o$ if they have pending events at the same configuration that contend on $o$.

**Property 1** (Disjoint access parallelism (weak))**.** *An STM implementation is* (weakly) disjoint-access parallel *if two transactions concurrently contend on the same base object only if they are not disjoint access.*

This definition captures the first condition of the disjoint-access parallelism property of Israeli and Rappoport [33], in accordance with most of the literature (cf. [30]). It is somewhat weaker, as it allows two processes to apply a trivial primitive on the same base object, e.g., read, even when executing disjoint-access transactions. Moreover, this definition only prohibits concurrent contending accesses, allowing transactions to contend on a base object $o$ at different points of the execution. A stronger requirement is:

**Property 2** (Disjoint access parallelism (strong))**.** *An STM implementation is* disjoint-access parallel *if two transactions concurrently access the same base object only if they are not disjoint access.*

The original disjoint-access parallelism definition [33] also restricts the impact of concurrent transactions on the *step complexity* of a transaction.

For additional definitions and discussion, see [6].

### 2.3.2 Invisibility of Reads

It is expected that many typical applications will generate workloads that include a significant portion of *read-only* transactions. This includes, for example, transactions to search a data structure, and find whether it contains a particular data item.

Many STMs attempt to optimize read-only transactions, and more generally, the implementation of read operations inside the transaction. By their very nature, read operations, and even more so, read-only transactions, need not leave a mark on the shared memory, and therefore, it is desirable to avoid writing in such transactions, i.e., to make sure that reads are *invisible*, and certainly, that read-only transactions do not write at all.

**Remark 2.** *Dice et al. [14] refer to a transaction as having invisible reads even if it writes, but the information is not sufficiently detailed to supply the exact details about the transaction's data set. (In their words, "the STM does not know which, or even how many, readers are accessing a given memory location.") This behavior is captured by the stronger notion of an* oblivious *STM [5].*

#### 2.3.3 Makespan Ratio

Some transactional memories come with a *scheduler*, determining which transaction to abort when there is a danger of violating consistency. One way to evaluate a transactional scheduler, borrowed from scheduling theory, is to measure its makespan, namely, the total time it takes to complete all the transactions in a specific workload.

Reducing the makespan is a major challenge, since transactions are often aborted and restarted. Measuring the makespan of a workload by itself is not indicative for the performance of a transactional scheduler, since the workload might be inherently sequential. Instead, the performance of a transactional scheduler is evaluated by the ratio, over all possible workloads, between its makespan and the makespan of an optimal, clairvoyant scheduler that knows the list of resource accesses that will be performed by each transaction, as well as its release time and duration [3, 21]. This idealistic transactional scheduler captures the inherent makespan needed to perform the workload, under complete knowledge, and the ratio captures the cost of the lack of this knowledge.

# 3 TM Lower Bounds and Impossibility Results

This section overviews research on the inherent complexity of TM. This includes several *impossibility results* showing that certain properties simply cannot be achieved by a TM, and a few *worst-case lower bounds* showing that other properties put a high price on the TM, often in terms of the number of steps that should be performed, or as bounds on the local computation involved.

## 3.1 Inherent Cost of TM Implementations

An early result demonstrates the additional cost of opacity over serializability, namely, the cost of making sure that the values read by a transaction are consistent as it is in progress (and not just at commit time, as done in many database implementations). Guerraoui and Kapałka [23] showed that the number of steps in a read operation is linear in the size of the invoking transaction's read set, assuming that reads are invisible, the STM keeps only a single version of each data item, and is progressive (i.e., it never aborts a transaction unless it conflicts with another pending transaction). In contrast, when only serializability has to be guaranteed, the values read can be validated only at commit time, leading to significant savings.

Another way to study the complexity of TM implementations is to prove lower bounds on objects that can be derived from them, for example, atomic snapshot objects [1]. Attiya et al. [2] have shown that if a wait-free implementation of an $m$-component snapshot object from historyless objects is space optimal, then its step complexity is in $\Omega(m)$. This follows from lower bounds for a new, more general class of implementations from base objects of any type.

Not all kinds of steps are created equal, and some steps involve more expensive synchronization than others, for example, those that force a memory barrier to occur. Kuznetsov and Ravi [35] have shown that if an STM implementation ensures a high degree of concurrency, i.e., it is permissive, then the number of expensive synchronization steps performed by a transaction is linear in its read-set size. The paper also demonstrates that only a constant number of synchronization steps is needed in each transaction of a strongly progressive STM; note that such STMs provide limited degree of concurrency. Nevertheless, even in strongly progressive STMs, a transaction must protect (e.g., by using locks or strong synchronization primitives) an amount of data that is linear in its write-set size.

## 3.2 The Consensus Number of TM

Consensus is a core problem in distributed computing, requiring processes to agree on one of their inputs. The *consensus number* of a data structure [27] is the maximal number of processes that can solve consensus using copies of the data structure (and read/write registers); the universality of consensus means that an object with consensus number $c$ can wait-free implement every other data structure, for $c$ processes, and that there are problems (specifically, consensus) that have no wait-free solution from the data structure (and read / write registers), for more than $c$ processes.

Guerraoui and Kapałka [22] have shown that lock-based and obstruction-free TMs can solve consensus for at most two processes, that is, their consensus number is 2. An intermediate step shows that such TMs are equivalent to shared objects that fail in a very clean manner [4]. Roughly speaking, this is a *consensus object* providing a familiar *propose* operation, allowing a thread to provide an input and wait for a unanimous decision value; however, the propose operation may return a definite *fail* indication, which ensures that the proposed value will not be decided upon. Intuitively, an aborted transaction corresponds to a propose operation returning false. To get the full result, further mechanisms are needed to handle the long-lived nature of transactional memory.

## 3.3 Providing Disjoint-Access Parallelism

Guerraoui and Kapałka [22] prove that obstruction-free implementations of software transactional memory cannot ensure *strict* disjoint-access parallelism. This property requires transactions with *disjoint data sets* (with strict disjoint access)

| | Strict DAP | Strong DAP | DAP |
|---|---|---|---|
| Opacity | Obstruction -freedom [22] | | |
| Linearizability | | | Wait-freedom [17] |
| Strict serializability | | | Invisible, wait -free reads [6] |
| Snapshot isolation | Obstruction -freedom [11] | Invisible, wait -free reads [6] | |

Table 1: Impossibility of achieving disjoint access parallelism (DAP). The table entry shows the progress condition needed for proving the result.

not to access a common base object. This notion is stronger than disjoint-access parallelism (Property 1), which allows two transactions with disjoint data sets to access the same base objects, provided they are connected via other transactions. Note that the lower bound does not hold under this more standard notion, as Herlihy et al. [28] present an obstruction-free and disjoint-access parallel STM.

The result that obstruction-free implementations of software transactional memory cannot ensure strict disjoint-access parallelism, has been extended in several important ways.

For the stronger case of wait-free read-only transactions, the assumption of strict disjoint-access parallel can be replaced with the assumption that read-only transactions are invisible. Specifically, an STM cannot be disjoint-access parallel and have invisible read-only transactions that always terminate successfully [6]. A read-only transaction not only has to write, but the number of writes is linear in the size of its read set. Both results hold for strict serializability, and hence also for opacity. With a slight modification of the notion of disjoint-access parallelism, i.e., strong disjoint-access parallelism (Property 2), these results also hold for serializability and snapshot isolation.

In fact, even the original result of Guerraoui and Kapałka [22] holds with snapshot isolation: Bushkov et al. [11] have shown that it is impossible to ensure strict disjoint-access parallelism and obstruction-freedom even if we weaken safety to ensure only snapshot isolation.

Another extension, by Ellen et al. [17], shows that transactional memory implementations cannot ensure both disjoint-access parallelism and wait-freedom; this assumes that the TM requires a process to re-execute its transaction if it has been aborted and that the TM guarantees that each transaction is aborted only a limited number of times.

Table 1 summarizes these impossibility results.

## 3.4 Privatization

An important goal for STM is to access certain items by simple reads and writes, without paying the overhead of the transactional memory. It has been shown [19] that, in many cases, this cannot be achieved without prior *privatization* [44, 45], namely, invoking a *privatization transaction*, or some other kind of a privatizing *barrier* [14].

Attiya and Hillel [5] have proved that, unless parallelism (in terms of progressiveness) is greatly compromised or detailed information about non-conflicting transactions is tracked (the STM is not *oblivious*), ensuring that no transaction writes to privatized data, incurs a cost (in terms of memory location accessed), which is linear in the number of items that are privatized.

## 3.5 Avoiding Aborts

An early result by Guerraoui et al. [20] shows that ensuring opacity, together with permissiveness is NP-hard. Similarly, Keidar and Perelman [34] prove that an opaque, strongly progressive STM requires NP-complete local computation, while a weaker, *online* notion requires visible reads.

The competitive ratio of the makespan is another way to measure the number of unnecessary aborts. It has been shown that the best competitive ratio achieved by simple transactional schedulers is $\Theta(s)$, where $s$ is the number of data items [3].

Attiya and Milani [7] studied the makespan of transactional scheduling under *read-dominated* workloads. These common workloads include *read-only* transactions, i.e., those that only observe data, and *late-write* transactions, i.e., those that update only towards the end of the transaction. This work shows that while read-only transactions are easily handled to achieve good makespan, late-write transactions significantly deteriorate the competitive ratio of any non-clairvoyant scheduler, assuming it takes a conservative approach to conflicts.

## 3.6 Limiting Progress

*Local progress* is a liveness property, which states that every process which is not parasitic (i.e., does not keep executing transactional operations without ever attempting to commit) and does not crash, makes progress. Bushkov et al. [12] defined this notion and proved that no TM implementation can ensure both opacity and local progress; in fact, the result holds also under the assumption of strict serializability.

Moreover, Crain et al. [13] have proved that opacity is incompatible even with *probabilistic* permissiveness, assuming reads are invisible. This means that there is no probabilistically permissive STM system that implements opacity while ensuring read invisibility. In contrast, probabilistic permissiveness can be obtained with the weaker condition, VWC [13].

**Acknowledgements:** The comments of Panagiota Fatourou, Petr Kuznetsov and Sandeep Hans helped to improve the presentation. The author was supported by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 238639, ITN project TRANSFORM.

# References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] Hagit Attiya, Faith Ellen, and Panagiota Fatourou. The complexity of updating snapshot objects. *Journal of Parallel and Distributed Computing*, 71(12):1570–1577, 2011.

[3] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. *Algorithmica*, 57(1):44–61, 2010.

[4] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *Journal of the ACM*, 56(4), 2009.

[5] Hagit Attiya and Eshcar Hillel. The cost of privatization. In *IEEE Transactions on Computers*, in press.

[6] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory Comput. Syst.*, 49(4):698–719, 2011.

[7] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *J. Parallel Distrib. Comput.*, 72(10):1386–1396, 2012.

[8] Hagit Attiya and Jennifer L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.

[9] G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA '93*, pages 261–270.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 1–10.

[11] Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. Snapshot isolation does not scale either. In *WTTM '13*.

[12] Victor Bushkov, Rachid Guerraoui, and Michał Kapałka. On the liveness of transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC'12)*, pages 9–18.

[13] Tyler Crain, Damien Imbs, and Michel Raynal. Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In *Algorithms and Architectures for Parallel Processing - 12th International Conference (ICA3PP'12)*, pages 244–257.

[14] Dave Dice, Alexander Matveev, and Nir Shavit. Implicit privatization using private transactions. In *5th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'10)*.

[15] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Distributed Computing, 20th International Symposium (DISC'06)*, pages 194–208.

[16] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

[17] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *ACM Symposium on Principles of Distributed Computing (PODC'12)*, pages 115–124.

[18] V. Gramoli, D. Harmanci, and P. Felber. Toward a theory of input acceptance for transactional memories. In *OPODIS '08*, pages 527–533.

[19] Rachid Guerraoui, Thomas Henzinger, Michał Kapałka, and Vasu Singh. Transactions in the jungle. In *Proceedings of the 22nd Annual Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, pages 263–272.

[20] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *Distributed Computing, 22nd International Symposium (DISC'08)*.

[21] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (PODC'05)*, pages 258–264.

[22] Rachid Guerraoui and Michał Kapałka. On obstruction-free transactions. In *SPAA '08*, pages 304–313.

[23] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 175–184.

[24] Rachid Guerraoui and Michał Kapałka. The semantics of progress in lock-based transactional memory. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 404–415.

[25] Rachid Guerraoui and Michał Kapałka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing. Morgan & Claypool Publishers, 2010.

[26] Sandeep Hans. Hagit Attiya. Transactions are back-but how different they are? TRANSACT, 2012.

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

[28] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 92–101.

[29] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*.

[30] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[31] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[32] Damien Imbs, Michel Raynal, and Jose Ramon de Mendivil. Brief announcement: virtual world consistency: a new condition for stm systems. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC'09)*, pages 280–281.

[33] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 151–160.

[34] Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. In *SPAA '09*, pages 59–68.

[35] Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *Principles of Distributed Systems - 15th International Conference (OPODIS'11)*, pages 112–127.

[36] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[37] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE transactions on computers*, 100(28):690–691, 1979.

[38] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

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

[40] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC'10)*, pages 16–25.

[41] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, pages 5–17.

[42] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*.

[43] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95*, pages 204–213.

[44] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. *SIGPLAN Not.*, 42(6):78–88, 2007.

[45] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. Technical Report Tr 915, Dept. of Computer Science, Univ. of Rochester, 2007.

[46] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'92)*, pages 212–222.

[47] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.