# The Distributed Computing Column

BY

**Stefan Schmid**

Aalborg University
Selma Lagerlöfs Vej 300, DK-9220 Aalborg, Denmark

The Distributed Computing Column features two articles: First, Michel Raynal takes us on a guided tour of distributed universal constructions. Subsequently, Srivatsan Ravi presents a survey of currently known complexity (upper and lower) bounds for implementing Transactional Memory as a shared object. Enjoy!

# Distributed Universal Constructions: a Guided Tour

Michel Raynal

Institut Universitaire de France
IRISA, Université de Rennes, 35042 Rennes, France
Department of Computing, Hong Kong Polytechnic University
raynal@irisa.fr

**Abstract**

The notion of a universal construction is central in computing science: the wheel has not to be reinvented for each new problem. In the context of $n$-process asynchronous distributed systems, a universal construction is an algorithm that is able to build any object defined by a sequential specification despite the occurrence of up to $(n-1)$ process crash failures. The aim of this paper is to present a guided tour of such universal constructions. Its spirit is not to be a catalog of the numerous constructions proposed so far, but a (as simple as possible) presentation of the basic concepts and mechanisms that constitute the basis these constructions rest on.

**Keywords**: Abortable object, Agreement problem, Asynchronous read/write system, Atomic operations, Computability, Concurrent object, Consensus, Crash failure, Disjoint-access parallelism, Help mechanism, LL/SC instruction, Memory location, Non-blocking, Obstruction-freedom, Progress condition, Sequential specification, $k$-Set agreement, $k$-Simultaneous consensus, Speculative execution, Universal construction, Wait-freedom.

## 1 Introduction

**A (very) short historical perspective** Looking for (some) universality seems inherent to humankind. Any language, any writing system, can be seen as an attempt to universality [42]. In the science domain, one of the very first witness of research of universality found in the past seems to be the Plimpton 322 tablet (Figure 1), which describes the fifteen first Pythagorean triplets ($a^2+b^2=c^2$).This

is only a list, not yet an algorithm with its proof. Hence, this tablet is a step to universality for Pythagorean triplets, but not yet a universal method able to provide us with a sequence of Pythagorean triplets of any length.

Figure 1: Plimpton 322 tablet

The geometric constructions with a compass and a straightedge designed by the Ancient Greeks are among the first algorithms coming with correctness proofs (see also [50]). Proofs of impossible constructions in the "compass + straightedge" computing model took more time (e.g., the impossibility of squaring the circle, i.e., build, with straightedge and compass only, a square whose area is equal to the area of a given circle)[1]. More recently, the Turing machine provides us with an abstract computing device, which is considered as the most general sequential computing model, thereby fixing the limits of what can be computed by a sequential machine [61][2]. It is consequently claimed to be *universal*. The *halting* problem is the most famous of the problems that are impossible to solve in this "most general" sequential computing model.

In distributed computing the situation is different. As written in [36]: "*In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine. In distributed systems, where computations require coordination among multiple par-*

[1]This impossibility follows from the fact that $\pi$ is a transcendent number (F. von Lindemann 1882), and a theorem by P. L. Wantzel, who established, in 1837, necessary and sufficient conditions for a number to be constructible in the "compass + straightedge" computing model [62].

[2]This means that any sequential computing model proposed so far has the same computability power as a Turing machine (e.g., Church's Lambda calculus, or Post systems [51]), or is weaker than a Turing machine (e.g., finite state automata).

*ticipants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants.*"

In distributed computing the main issues posed by universality and computability appear when one has to implement distributed state machines (distributed services encapsulated in concurrent objects) in the presence of adversaries due to the environment in which the computation evolves (such as asynchrony and process failures) [25, 32, 43, 46].

**Concurrent objects and asynchronous crash-prone read/write systems** A concurrent object is an object that can be accessed (possibly simultaneously) by several processes. From both practical and theoretical point of views, a fundamental problem of concurrent programming consists in implementing high level concurrent objects, where "high level" means that the object provides the processes with an abstraction level higher than the atomic hardware-provided instructions. While this is well-known and well-mastered since a long time in the context of failure-free systems [13], it is far from being trivial in failure-prone systems (e.g., see textbooks such as [52, 58]), where it is still an important research domain.

This paper considers systems made up of $n$ sequential asynchronous processes which, at the hardware level, communicate through memory locations (memory words also called registers) which can be accessed by atomic operations (instructions), including the basic read and write operations. Moreover, it is assumed that, in any run, up to $(n-1)$ processes may crash (unexpected halting). When restricted to the basic read and write instructions, this computation model is known under the name *wait-free read/write* model (denoted here $\mathcal{CARW}_n[\emptyset]$, where $\mathcal{CARW}$ stands for Crash Asynchronous Read/Write).

**On progress conditions** Deadlock-freedom and starvation-freedom are well-known progress conditions in failure-free asynchronous systems. As their implementation is based on lock mechanisms, they are not suited to asynchronous crash-prone systems. This is due to the fact that it is impossible to distinguish a crashed process from a slow process, and consequently a process that acquires a lock and crashes before releasing it can entail the blocking of the entire system.

Hence, new progress conditions for concurrent objects suited to crash-prone asynchronous systems have been proposed. Given an object, we have the following.

- The strongest progress condition is *wait-freedom* (WF) [32]. It states that, any operation (on the object that is built) issued by a process that does not crash terminates. This means that it terminates whatever the behavior of the

other processes. This can be seen as the equivalent of the starvation-freedom progress condition encountered in failure-free systems.

- The *non-blocking* progress condition (NB) states that there is at least one process that can always progress (all its object operations terminate) [38]. This progress condition is also called *lock-freedom*. It can be seen as the equivalent of deadlock-freedom in failure-free systems. Non-blocking has been generalized in [14], under the name *k-lock-freedom* ($k$-NB), which states that at least $k$ processes can always make progress.
- The *obstruction-freedom* progress condition (OB) states that a process that does not crash will be able to terminate its operation if all the other processes hold still long enough [34]. This is the weakest progress condition. It has been generalized in [59], under the name *k-obstruction-freedom* ($k$-OB), which states that, if a set of at most $k$ processes run alone for a sufficiently long period of time, they will terminate their operations.

While *wait-freedom* and *non-blocking* are independent of the concurrency and failure pattern, *obstruction-freedom* is dependent from it. Asymmetric progress conditions have been introduced in [41]. The computational structure of progress conditions is investigated in [60].

**Universal construction** The notion of a universal construction was introduced by M. Herlihy in [32]. It considers objects (a) which are defined from sequential specifications and (b) whose operations are total, i.e., any object operation returns a result (as an example, a push() operation on an empty stack returns the default value $\perp$).

Let $PC$ be a progress condition. A *PC-compliant universal construction* is an algorithm that, given the sequential specification of an object $O$ (or a sequential implementation of it), provides a concurrent implementation of $O$ satisfying the progress condition $PC$ in the the presence of up $(n-1)$ process crashes (Figure 2).

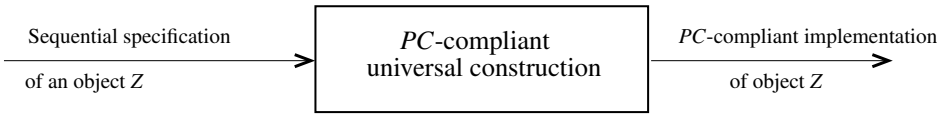



Figure 2: $PC$-compliant universal construction

It has been shown in [25, 32, 47] that the design of a universal construction with respect to the wait-freedom progress condition is impossible in $\mathcal{CARW}_n[\emptyset]$. This means that the basic system model $\mathcal{CARW}_n[\emptyset]$ has to be enriched with hardware-provided atomic instructions or additional computing objects whose computational power is stronger than atomic read/write registers (in the following,

we consider terms "register" and "memory location" as synonyms; we sometimes also say "atomic read/write object" by a slight abuse of language).

**Content of the paper** This paper aims at being a guided tour to distributed universal constructions. Its goal is not to be a presentation including as many universal constructions as possible, but to focus on the central features universal constructions rest on, and illustrate them with existing algorithms. To this end, after having introduced basic definitions (Section 2), the paper proceeds as follows.

- Section 3 presents first a simple and elegant universal construction suited to the system model $\mathcal{CARW}_n$[LL/SC] (which is $\mathcal{CARW}_n[\emptyset]$ enriched with the hardware-provided instructions LL and SC, which are defined in the section). This allows for an easy introduction of the notion of a *speculative computation* and the notion of a *help mechanism* (introduced in [32] and recently formalized in [17]). This section presents also extensions devoted to *large* objects.
- Section 4 is made up of two subsections. the first is on the efficiency of universal constructions. Considering the algorithms that realize them, it addresses the notion of *disjoint-access parallelism*.

  The second subsection is on the object side. It considers the case of universal constructions for deterministic *abortable* objects [15, 31, 52, 53]. Such an object is a classical object defined by a sequential specification which allows an operation to return a default value $\perp$ in the presence of contention (in this case the operation has no effect on the object). Hence, in a concurrency-free execution, an abortable object behaves as its non-abortable counterpart. The notion of $k$-abortable object has been recently introduced in [8], where is also presented an associated universal construction. A $k$-abortable object is such that an operation is allowed to return $\perp$ only if it is concurrent with operations from at most $k$ different processes, and none these operations return $\perp$.
- All the previous universal constructions consider that the underlying crash-prone system is enriched with hardware-provided atomic instructions such as LL/SC or Compare&Swap, which work on memory locations [22]. Hence, the question: Which are the instructions that allow to build a universal construction? As an example, can a universal construction be designed for the system model $\mathcal{CARW}_n$[Test&Set] ($\mathcal{CARW}_n[\emptyset]$ enriched with the hardware-provided atomic instruction Test&Set). This issue was solved by M. Herlihy in [32], who introduced the celebrated *consensus hierarchy*. This is addressed in the first part of Section 5. Hence, the consensus object is at the core of universal constructions.

Then, the section shows another important advantage of using consensus objects instead of primitives hardware-provided instructions to design universal constructions. While instructions are uniform (any instruction can access any memory location [22]), an object is a typed abstraction that has the property that an operation on type $T$ cannot be applied to an object of type $T'$. Moreover, an object can be weakened or generalized according to the needs of the user. As an example, the consensus object can be weakened to the $k$-set agreement ($k$-SA) object [19] or to the $k$-simultaneous consensus ($k$-SC) object [3] ($k$-SA and $k$-SC objects are defined in the section).

The section presents then the notion of a $k$-universal construction due to E. Gafni and R. Guerraoui [27]. Such a construction considers $k$ objects (instead of only one) and ensures that at least one of these objects progresses forever. This construction relies on $k$-SC objects instead of consensus objects.

Finally, the section considers the case where we want that, not at least one but at least $\ell$ objects progress forever, where $\ell$ is any predefined constant in $[1..k]$. As shown in [55], objects denoted $(k,\ell)$-SC ($(k,\ell)$-simultaneous consensus objects defined in the section), which are strictly stronger than $k$-SC objects (when $\ell > 1$), and weaker than consensus objects (when $\ell < k$), are necessary and sufficient to build a universal construction for $k$ objects, where at least $\ell$ objects progress forever. It is important to notice that these generalizations of universal constructions could not have been obtained from hardware-provided instructions. This will conclude the guided tour.

Finally, after a short Section 6 comparing universal constructions and software transactional memory (STM) systems, Section 7 concludes the paper.

# 2 Basic Asynchronous Read/Write Model $\mathcal{CARW}_n[\emptyset]$

**Crash-prone asynchronous processes** The basic computing model (denoted here $\mathcal{CARW}_n[\emptyset]$) was sketched in the introduction. It is composed of a set of $n$ sequential processes denoted $p_1$, ..., $p_n$. Each process is asynchronous which means that it proceeds at its own speed, which can be arbitrary and remains always unknown to the other processes.

A process may halt prematurely (crash failure), but executes correctly its local algorithm until it possibly crashes. Up to $(n-1)$ processes may crash in a run. Due to the atomicity of the hardware-provided operations, if a process crashes while executing such an operation, this operation appears as entirely executed or not at all. A process that crashes in a run is said to be *faulty* in this run. Otherwise, it

is *correct* or *non-faulty*. Hence, a faulty process is a process whose speed, after some time, remains forever equal to 0.

**On atomicity** The processes communicate by accessing atomic read/write registers (memory locations). Atomicity means that the read and write primitive operations on a register appear as if they have been executed one after the other. Moreover, the corresponding sequence of operations $S$ is such that (a) if the operation $\mathsf{op}_1$ terminated before the operation $\mathsf{op}_2$ started, $\mathsf{op}_1$ appears before $\mathsf{op}_2$ in $S$, and (b) a read operation on a register $R$ returns the value written by the closest preceding write operation on $R$ (or its initial value if there is no preceding write) [44]. Atomicity is also called *linearizability* when considering any object defined by a sequential specification [38].

**Notation** Variables local to a process $p_i$ are denoted with lowercase letters, sometimes indexed with $i$. Memory location and objects shared by the processes are denoted with capital letters.

# 3 A Simple LL/SC-Based WF-Compliant Universal Construction

## 3.1 Extending $\mathcal{CARW}_n[\emptyset]$ with LL/SC

**Model $\mathcal{CARW}_n$[LL/SC]** These hardware-provided atomic instructions can be applied to any memory location. The wait-free read/write model $\mathcal{CARW}_n[\emptyset]$ enriched with them is denoted $\mathcal{CARW}_n$[LL/SC]. LL/SC is made up of three instructions: LL stands for Linked Load; SC stands for store conditional; VL stands for Validate.

Let $X$ be a memory location. $X.\mathsf{LL}()$ returns the current value of $X$. Let $p_i$ be a process that invokes $X.\mathsf{SC}(v)$. This invocation assigns $v$ to $X$ if $X$ has not been assigned a value by another process since the previous invocation of $X.\mathsf{LL}()$ issued by $p_i$. In this case, $X.\mathsf{SC}(v)$ returns `true` and we say that the invocation is successful; otherwise it returns `false`. Finally, an invocation of $X.\mathsf{VL}()$ by process $p_i$ returns `true` if no other process has issued a successful $X.\mathsf{SC}()$ since the last invocation of $X.\mathsf{LL}()$ issued by $p_i$.

These instructions are used to bracket a *speculative computation*. A process first reads $X$ with $X.\mathsf{LL}()$ and stores its value in a local variable $x_i$. Then $p_i$ does a local computation which depends on both $x_i$ and its local state. The aim of this local computation is to define a new value $v$ for $X$. Finally, $p_i$ tries to commit its local computation by writing $v$ into $X$, which is done by invoking $X.\mathsf{SC}(v)$.

If this invocation is successful, the write is committed; otherwise the write fails. A similar behavior can be obtained by the Compare&Swap() instruction. The main advantage of LL/SC, with respect to Compare&Swap(), lies in the fact that it does suffer the ABA problem (see [52, 58]), which requires sequence numbers to be solved. Algorithms based on LL/SC can be found in many publications (e.g., [23, 33, 39, 52, 58, 59] to cite a few).

## 3.2 A simple universal construction in $\mathcal{CARW}_n$[LL/SC]

This section presents a simplified version (denoted sFK) of a universal construction due to P. Fatourou and N. Kallimanis [24]. The main difference is that the presented construction uses sequence numbers which increase forever, while [24] uses sequence numbers modulo 2). This additional memory cost makes it much easier to present and prove correct.

**Collect object** This construction uses a collect object. Such an object can easily be built in $\mathcal{CARW}_n[\emptyset]$. It consists of an array $COL[1..n]$, with one entry per process, and provides them with two operations denoted update() and collect(). The invocation of $COL$.update($v$) by a process $p_i$ assigns $v$ to $COL[i]$. The invocation of $COL$.collect() is an asynchronous scan of the array which returns, for each entry $j$, the value it has read from $COL[j]$. A formal definition of such an object can be found in [52].

Due to the asynchronous scan, a collect object is not atomic (hence a collect object is computationally weaker than a snapshot object [1]). An atomic version of a collect object is described in [24]. Its implementation (a) assumes that the $n$ components of the collect object are stored in a single memory location, and (b) is based on the hardware-provided instruction add() ($Y$.add(v) atomically adds $v$ to $Y$).

**Global and local variables** Let $O$ be the object that is built.

- $STATE$ is a memory location made up of three fields:
  - $STATE.value$ contains the current value of $O$. It is initialized to the initial value of $O$.
  - $STATE.sn[1..n]$ is an array of sequence numbers initialized to $[0, \cdots, 0]$; $STATE.sn[i]$ is the sequence number of the last invocation of an operation on $O$ issued by $p_i$.
  - $STATE.res[1..n]$ is an array of result values initialized to $[\perp, \cdots, \perp]$; $STATE.res[i]$ contains the result of the last operation issued by $p_i$ that has been applied to $O$.

- *BOARD* is a collect object. $BOARD[i]$ is a pair $\langle BOARD[i].op, BOARD[i].sn\rangle$ initialized to $\langle \perp, 0\rangle$; $BOARD[i].op$ contains the last operation on $O$ issued by $p_i$, and $BOARD[i].sn$ contains its sequence number.
- Each process $p_i$ manages a sequence number generator $sn_i$ initialized to 1.

The object $O$ is assumed to be defined by a transition function $\delta()$. Let $s$ be the current state of $O$ and op($in$) be the invocation of the operation op() on $O$, with input parameter $in$; $\delta(s, \mathsf{op}(in))$ outputs a pair $\langle s', r\rangle$ such that $s'$ is the state of $O$ after the execution of op($in$) on $s$, and $r$ is the result of op($in$).

**Construction sFK: speculative computation and helping** The construction sFK is described in Figure 3. When a process $p_i$ invokes an operation op($in$) on $O$, it first publishes the pair $\langle \mathsf{op}(in), sn_i\rangle$ in the collect object *BOARD* (line 1). Then, it invokes the internal procedure apply() at the end of which it will locally return the result produced by op($in$) (line 2).

```
when p_i invokes op(in) do
(1)  BOARD.update(⟨op(in), sn_i⟩); sn_i ← sn_i + 1;
(2)  apply(); let r = STATE.res[i]; return(r).

internal procedure apply() is
(3)  repeat twice
(4)      ls ← STATE.LL();
(5)      pairs ← BOARD.collect();
(6)      for ℓ ∈ {1, 2, ··· , n} do
(7)          if (pairs[ℓ].sn = ls.sn[ℓ] + 1 then
(8)              ⟨new_state, r⟩ ← δ(ls.value, pairs[ℓ].op);
(9)              ls.res[ℓ] ← r; ls.sn[ℓ] ← pairs[ℓ].sn
(10)         end if
(11)     end for
(12)     STATE.SC(ls)
(13) end repeat twice.
```

Figure 3: WF-compliant universal construction sFK (system model $\mathcal{CARW}_n$[LL/SC])

The core of the construction is the procedure apply(), in which a process $p_i$ executes twice the lines 4-12 (we will see later why this has to be done twice). Process $p_i$ first reads the current local state of the object (line 4), and starts a first speculative execution (which will end at line 12). In this speculative execution, $p_i$ first reads the content of the collect object *BOARD* from which it obtains for each process $p_\ell$ a pair ⟨last operation invoked by $p_\ell$, associated sequence number⟩. Let us recall that as *BOARD*.collect() is not atomic, and $p_i$ is asynchronous, the pairs

that are returned are not necessarily associated with a consistent global state the computation passed through.

Then, $p_i$ considers each pair in *pairs* in the "for" loop of lines 6-11. In this loop, $p_i$ strives to help all the processes that have a pending operation on $O$. From its point of view (i.e., with the information it has obtained from its previous reads of *STATE* and *BOARD*), those are all the processes $p_\ell$ such that $pairs[\ell].sn = ls.sn[\ell] + 1$ (line 7). If this local predicate is true, $p_i$ locally simulates (speculative computation) the last operation issued by $p_\ell$ not yet applied to the object (line 6), and locally saves the result of the operation and its sequence number (line 9). Finally, $p_i$ tries to commit its speculative computation by invoking *STATE*.SC() (line 12). Let us observe that, if this invocation is successful, we can conclude that no process modified *STATE* while $p_i$ was doing its speculative computation. Hence, the local variable $ls$ of $p_i$ is up to date, and, from an external observer point of view, everything appears as if the computation starting at line 4 and ending at line 12 was executed atomically. If the invocation of *STATE*.SC() is not successful, the speculative execution is not committed.

**Construction sFK: why "repeat twice"?** Let us first observe that, due to sequence numbers, once registered in the collect object *BOARD*, an operation cannot be executed more than once. Moreover, if the process $p_i$ that invokes an operation does not crash, it terminates its operation op($in$). This follows from the fact that the lines 7-10 are executed a bounded number of times ($2n$). But is the result provided for op($in$) correct?

To answer this question, let us consider the execution described in Figure 4. When process $p_j$ (bottom of the figure) executes the atomic statement *STATE*.LL() followed by *BOARD*.collect() (lines 4-5), $p_i$ (top of the figure) has not yet registered by executing *BOARD*.update() (line 1). Hence $pairs_j$ does not contain $\langle$op($in$), $sn\rangle$. Let us assume that the execution of *STATE*.SC($ls_j$) by $p_j$ is successful. If $p_i$ executes only once the repeat loop, its execution of *STATE*.SC() is not successful, and $p_i$ returns despite the fact that $p_j$ has not helped it by executing op($in$). Hence, the statement return($r$) executed by $p_i$ at line 2 returns the result of its previous operation invocation.

Assuming now that $p_i$ executes twice the repeat loop, let us consider the first successful invocation of *STATE*.SC() that occurs after the previous successful invocation by $p_j$. This invocation is issued by some process $p_k$ (which can be $p_i$, $p_j$ or any other process). According to the algorithm of Figure 3, it follows that $p_k$ has previously invoked *STATE*.LL(). Moreover, this invocation occurs necessarily after the successful invocation of *STATE*.SC() by $p_j$ (otherwise the invocation of *STATE*.SC() by $p_k$ could not be successful). Consequently, the invocation of *BOARD*.collect() by $p_k$ is such that $\langle$op($in$), $sn\rangle \in pairs_k$. It follows that $p_k$

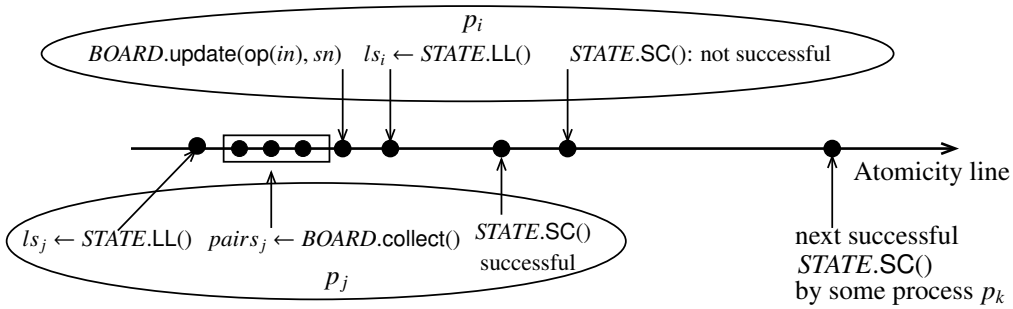



Figure 4: Why to repeat twice lines 4-12 (big dot = atomic step)

found $pairs_k[i].sn = ls_k.sn[i] + 1$, and simulated the execution of op($in$) on behalf of $p_i$ and wrote the corresponding result in $ls_k.res[i]$ which was then copied in $STATE.res[i]$ by the successful execution of $STATE$.SC() by $p_k$.

**Linearization of the operations on $O$** Let SC[1], SC[2], ..., SC[$x$], etc. be the sequence of all the successful invocations of $STATE$.SC(); as $STATE$.SC() is atomic, this sequence is well-defined. Starting from $SC[1]$, each SC[$x$] applies at least one operation on the object $O$. It is possible to totally order the operations applied to $O$ by each SC[$x$]. Let seq[$x$] be the corresponding sequence. The sequence of operations applied to $O$ is then seq[1] followed by seq[2], ..., followed by seq[$x$], etc.

**Remark on sequence numbers** Techniques such as the one described in [9, 48] (known under the name *alternating bit protocol*) can be used to obtain an implementation in which the sequence numbers are implemented modulo 2.

## 3.3 The case of large objects

The previous universal construction considered that the internal state of the object ($STATE$) can be copied all at once. A *large* object is an object whose internal state cannot be copied in one instruction.

Several articles have addressed this problem, e.g., [2, 6, 33]. They all propose to fragment a large object into blocks. Two main approaches have been proposed.

- One consists in using pointers linking the blocks representing the object [33]. Moreover, it requires that the programmer provides a sequential implementation of the object that performs as little copying as possible. The pointers are then accessed with LL instructions which allow a process to obtain a logical copy of the object (which means that only the needed part of the object is copied in its local memory). A process executes then locally a

speculative computation, as defined by the operation it wants to apply to the object. Finally it uses SC instructions on the appropriate pointers to try to commit the new value of the object.

- The other approach consists in representing the object as a long array fragmented into blocks [6]. This paper presents two object constructions based on this approach, which are universal with respect to non-blocking and wait-freedom, respectively. It also presents algorithms implementing atomic LLL/LSC operations (where "L" stands for Large), which extend the LL/SC instructions to arrays of memory locations. These operations are built in the system model $\mathcal{CARW}_n$[LL/SC].

# 4 Extensions

This section presents two extensions of universal constructions. The first one regards their efficiency. The second one considers a weakening of concurrent objects called abortable objects.

## 4.1 On the implementation side: Disjoint-access parallelism

**Disjoint-access parallelism** A universal construction is *disjoint-access parallel* if two processes that access distinct parts of an object $O$ do not access common base objects or common memory location which constitute the internal representation of $O$. As an example, let us consider a queue. If the queue contains three or more items, a process executing enqueue($v$) and a process executing dequeue() must be able to progress without interfering.

Hence, the aim of a disjoint-access parallel universal construction is to provide efficient implementations. Let us observe that all the universal constructions that built a total order on the operations (such as the one described in Section 3.2 and the ones presented in [2, 23, 33]) are not disjoint-access parallel.

**What can be done?** Hence the question posed by F. Ellen, P. Fatourou, N. Kosmas, A. Milani, and C. Travers, in [21]: Is it possible to design a disjoint-access parallel WF-compliant universal construction? This work presents two important results.

- The first is an impossibility result. It states that it is impossible to design a universal construction that is disjoint-access parallel and ensures that all the operation invocations of the processes that do not crash always terminate. Hence, when we consider any object defined by a sequential specification, disjoint-access parallelism and wait-freedom are mutually exclusive.

- The second result is a positive one, namely the previous impossibility (which considers *any* object defined by a sequential specification) does not apply to a special class of concurrent objects. Hence, the constructions for this object class are no longer "universal" in the strict sense. This object class contains all the objects $O$ for which, in any sequential execution, each operation accesses a bounded number of base objects used to represent $O$. Examples of such objects are bounded trees, or stacks and queues whose internal representations are list-based.

  In their paper, the authors describe a universal construction that ensures, for the previous objects, both the disjoint-access parallel property of the object implementation, and the wait-freedom progress condition for the processes that use it. This construction is presented in the system model $\mathcal{CARW}_n$[LL/SC].

## 4.2 On the object side: Abortable objects

Abortable objects have been investigated in several articles, e.g., [4, 15, 31, 52, 53]. They found their origin in the commit/abort output of transaction-based systems [28], and the notion of "fast path" initially introduced in fast mutual exclusion algorithms [45].

**Definition** An abortable object is an object (defined by a sequential specification) such that

- When executed in a contention-free context, an operation takes effect, i.e., modifies the state of the object and returns a result as defined by its sequential specification,

- When executed in a contention context, an operation either takes effect and returns a result as defined by its sequential specification, or returns the default value $\perp$ (abort). If $\perp$ is returned, the operation has no effect on the state of the object.

Hence, an abortable object is such that any operation always returns (i.e., whatever the concurrency context). Its progress condition is consequently wait-freedom. Differently from an abortable object, an obstruction-free object does not guarantee operation termination in the presence of concurrency. A theory of deterministic abortable objects (including a study of their respective power) is presented in [31].

**Universal constructions for abortable objects** Such a very simple construction is described in Figure 5. It is a trivial simplification of the universal construction described in Figure 3 from which the helping mechanism has been suppressed. The memory location *STATE* contains now only the state of the object.

```
when p_i invokes op(in) do
(1) ls ← STATE.LL();
(2) ⟨new_state, r⟩ ← δ(ls, pairs[ℓ].op);
(3) done ← STATE.SC(ls);
(4) if (done) then return(r) else return(⊥) end if.
```

Figure 5: WF-compliant universal construction for abortable objects (system model $\mathcal{CARW}_n$[LL/SC])

When a process $p_i$ invokes an operation op($in$) on the object, it reads its current state to obtain a local copy (line 1). Then it produces a speculative execution of op($in$) on this local state $ls$ (line 2). Finally, it tries to commit its local execution by issuing *STATE*.SC($ls$) (line 3). If this SC is successful, $p_i$ returns the result it has previously computed. Otherwise, there was at least one concurrent operation, and $p_i$ returns $\perp$ (line 4).

Let us observe that, if several processes concurrently invoke operations, each invokes *STATE*.LL(), and the first of them that invokes *STATE*.SC() produces a successful SC. It follows that, in the presence of concurrency, at least one process is guaranteed to make progress in the sense that it does not return $\perp$.

An efficient *solo-fast* universal construction for deterministic abortable objects is described in [15]. Solo-fast (also called contention-aware in other articles) means that the implementation is allowed to use atomic operations on memory locations stronger than read/write only when there is contention. Moreover, this implementation guarantees that the operations that do not modify the object never return $\perp$ and use only read/write operations. This implementation is based on the primitive operation on memory locations Compare&Swap, whose computational power is the same as LL/SC.

**$k$-Abortable objects** This notion was recently introduced in [8]. A *$k$-abortable* object guarantees progress even under high contention, where "progress" means that $\perp$ cannot be returned by some operation invocations.

Roughly speaking an operation invoked by a process is allowed to abort only if it is concurrent with operations issued by $k$ distinct processes and none of them returns $\perp$. This means that the $k$ operations that entail the abort of another operation must succeed. It is easy to see that $n$-abortability is wait-freedom where any operation returns a non-$\perp$ result. A formal presentation can be found in [8].

A universal construction for $k$-abortable objects suited to the system model $\mathcal{CARW}_n$[LL/SC] is presented in [8]. Differently from the trivial construction for abortable objects presented in Figure 5, it is not a trivial construction. It uses an array of $n$ memory locations $BOARD[1..n]$ used by the processes to store their last operations (they are the equivalent of the collect object $BOARD[1..n]$ used in Figure 3), an array of $k$ memory locations $WINNERS[1..k]$ which contains the (up to $k$) "winning" operations, and another memory location $STATE$ (similar to the location $STATE$ used in Figure 3). All these memory locations are accessed with the LL/SC atomic operations. (We use the same identifiers as in Figure 3 to facilitate the understanding.)

The construction works as follows. After it has registered its operation in $BOARD[i]$, a process $p_i$ tries to find an available entry in $WINNERS[1..k]$. If it succeeds, its operation will not abort; otherwise its operation will eventually abort. In all cases, i.e., whatever the fate of its own operation, the process $p_i$ will help the winning operations to terminate. This construction is efficient in the sense that each operation terminates in $O(k)$ accesses to memory locations.

Let us observe that, as every $k$-abortable object can easily implement its $k$-lock-free counterpart, the previous universal construction for $k$-abortable objects is $k$-NB-compliant universal construction. Let us remember that, differently from its $k$-lock-free counterpart, no process can get stuck when a $k$-abortable object is used.)

# 5 From Operations on Memory Locations to Agreement Objects

## 5.1 Primitive operations versus objects

The previous universal constructions are based on hardware-provided atomic operations such as LL/SC. This operation, as all the hardware-provided synchronization operations (such as Test&Set or Compare&Swap) is uniform in the sense that they can be applied to any memory location [6, 22]. Hence the following natural questions come to mind:

- Is it possible to design a universal construction with other hardware-provided atomic operations such as Test&Set or Fetch&Add, initially designed to solve synchronization issues? Moreover, which synchronization atomic operations are equivalent (from the point of view of a universal construction)?

- Is it possible to generalize the concept of a universal construction to the coordinated construction of several objects with different progress conditions?

These questions are answered in this section.

## 5.2 A fundamental agreement object: consensus

Differently from a memory location which is only a sequence of bits accessed by hardware-provided atomic operations, the aim of an object is to provide its user with a high abstraction level (by hiding implementation details) and allow easier reasoning and proofs. An object is defined by a set of operations, and a specification which describes its correct behavior. The operations associated with an object are specific to it (i.e., due the very essence of the object concept, they are not uniform).

**The consensus object** The consensus object is the fundamental object associated with agreement problems. Introduced (in a different form) in the context of Byzantine synchronous message-passing systems [46], a consensus object provides the processes with a single operation denoted propose() that a process can invoke only once (one-shot object). When a process invokes propose($v$), we say that it "proposes the value $v$". This operation returns a result. If a process returns value $w$, we say that it "decides $w$". In the context of process crash failures, the consensus object is defined by the following set of properties (let us recall that a correct process is a process that does not crash).

- Termination. If a correct process invokes propose(), it decides a value.
- Validity. A decided value is a proposed value.
- Agreement. No two processes decide different values.

A consensus object allows the processes to agree on the same value, and this value is not arbitrary: it was proposed by one of them. Hence, when considering a universal construction, consensus objects can be used by the processes to agree on the order in which their operations must be applied to the object that is built.

## 5.3 A simple consensus-based universal construction

A simple WF-compliant consensus-based universal construction is described in Figure 6. This construction, proposed in [30], is inspired from the state machine replication paradigm [43] and the consensus-based atomic broadcast algorithm presented in [18]. The reader will find a proof of it in [52]. Let $O$ be the object that is built. As in Section 3, its sequential behavior is defined by a transition function $\delta()$.

**Local variables** A process $p_i$ manages locally a copy of the object, denoted $state_i$, an array $sn_i[1..n]$ where $sn_i[j]$ denotes the sequence number of the last operation on $O$ issued by $p_j$ locally applied to $state_i$. The local variables $done_i$, $res_i$, $prop_i$, $k_i$, and $list_i$, are auxiliary variables whose meaning is clear from the context; $list_i$ is a list of pairs of (operation, process identity); $|list_i|$ is its size, and $list_i[r]$ is its $r^{\text{th}}$ element; hence, $list_i[r].op$ is an object operation and $list_i[r].proc$ the process that issued it.

```
when p_i invokes op(in) do
(1)   done_i ← false; BOARD[i] ← ⟨op(in), sn_i[i] + 1⟩;
(2)   wait (done_i); return(res_i).

Underlying local task T:    % background server task %
(3)   while (true) do
(4)       prop_i ← ϵ;   % empty list %
(5)       for j ∈ {1, ..., n} do
(6)           if (BOARD[j].sn > sn_i[j]) then
(7)               append (BOARD[j].op, j) to prop_i
(8)           end if
(9)       end for;
(10)      if (prop_i ≠ ϵ) then
(11)          k_i ← k_i + 1;
(12)          list_i ← CONS[k_i].propose(prop_i);
(13)          for r = 1 to |list_i| do
(14)              ⟨state_i, res_i⟩ ← δ(state_i, list_i[r].op);
(15)              let j = list_i[r].proc; sn_i[j] ← sn_i[j] + 1;
(16)              if (i = j) then done_i ← true end if
(17)          end for
(18)      end if
(19)  end while.
```

Figure 6: A wait-free consensus-based universal construction (code for process $p_i$)

**Shared Objects** The shared memory contains the following objects.

- An array $BOARD[1..n]$ of single-writer/multi-reader atomic registers. Each entry is a pair such that the pair $\langle BOARD[j].op, BOARD[j].sn\rangle$ contains the last operation issued by $p_j$ and its sequence number. Each $BOARD[j]$ is initialized to $\langle\perp, 0\rangle$.
- An unbounded array $CONS[1..]$ of consensus objects.

**Process behavior** When a process $p_i$ invokes an operation op($in$) on $O$, it registers it and its associated sequence number in $BOARD[i]$ (line 1). Then, it waits until the operation has been executed, and returns its result (line 2).

The array *BOARD* constitutes the helping mechanism used by the background task of each process $p_i$. This task is made up two parts, which are repeated forever. First, $p_i$ build a proposal $prop_i$, which includes the last operations (at most one per process) not yet applied to the object $O$, from its local point of view (lines 4-9 and predicate of line 6). Then, if the sequence $prop_i$ is not empty, $p_i$ proposes it to the next consensus instance $CONS[k_i]$ line 12). The resulting value $list_i$ is a sequence of operations proposed by a process to this consensus instance. Process $p_i$ then applies this sequence of operations to its local copy $state_i$ of $O$ (line 14), and updates accordingly its local array $sn_i$ (line 15). If the operation that was applied is its own operation, $p_i$ sets the Boolean $done_i$ to true (line 16), which will terminate its current invocation (line 2).

**Bounded wait-freedom versus unbounded wait-freedom** This construction ensures that the operations issued by the processes are wait-free, but does not guarantee that they are bounded-wait-free, namely, the number of steps (accesses to the shared memory) executed before an operation terminates is finite but not bounded. Consider a process $p_i$ that issues an operation $\mathsf{op}()$, while $k1$ is the value of $k_i$. let and $k2 = k1 + \alpha$ be such that $\mathsf{op}()$ is output by the consensus instance $CONS[k2]$. The task $T$ of $p_i$ must execute $\alpha$ times the lines 4-18 in order to catch up the consensus instance $CONS[k2]$ and return the result produced by $\mathsf{op}()$. It is easy to see that the quantity $(k2 - k1)$ is always finite but cannot be bounded.

A bounded construction is described in [32]. Instead of requiring each process to manage a local copy of the object, $O$ is kept in shared memory and is represented by a list of cells including an operation, the resulting state, the result produced by this operation, and a consensus object whose value is a pointer to the next cell. The last cell defines the current value of the object.

## 5.4 Consensus number and the consensus hierarchy

**Consensus number of an object** The notion of the *consensus number* of an object was introduced by M. Herlihy in [32]. Let us consider an object of type $T$ (defined by a sequential specification). The *consensus number* of an object of type $T$ is the greatest integer $n$ such that it is possible to implement a consensus object in a system of $n$ processes, with any number of atomic read/write registers and objects of type $T$. The consensus number is $+\infty$ if there is no largest $n$.

This notion allows us to answer the first question posed in Section 5.1, and this answer defines what is called the object *consensus hierarchy*. More precisely, it has been shown in [32] that:

- The consensus number of read/write registers is 1. It follows that all objects that can be built from read/write registers only (i.e., in $\mathcal{CARW}_n[\emptyset]$ without

enrichment with additional operations) have consensus number 1. Snapshot objects [1, 5] and renaming objects [7, 16] are such objects.

- The consensus number of hardware operations such as Test&Set, Fetch&Add, Swap (exchange the values in a local register an a shared register), and a few others, have consensus number 2. This means that a universal construction can be built in $\mathcal{CARW}_2$[Test&Set] (i.e., in a system of two processes), but impossible in $\mathcal{CARW}_n$[Test&Set] for $n > 2$.
- Let a $k$-window read/write register be a register that stores only the sequence of the last $k$ values which have been written, and whose read operation returns this sequence of at most $k$ values. It is shown in [49] that the consensus number of a $k$-window is $k$.
- Finally, the consensus number of Compare&Swap, LL/SC, and a few others, is $+\infty$.

This infinite hierarchy is the *consensus hierarchy*. It provides us with a ranking of the power of synchronization objects and hardware provided synchronization operations in wait-free systems (i.e., systems where all, except one, processes may crash). As an example, if any number of processors may crash, this hierarchy states that a multicore with Test&Set is computationally less powerful than a multicore with LL/SC.

**Consensus from several operations on memory locations** The previous hierarchy considers that consensus must be built from read/write registers and objects of a given type $T$ only. What can be done when several hardware operations which access the same memory locations are given?

As an example, let $\mathcal{CARW}_n$[Test&Set, Fetch&Add2] be the system model (defined in [22]) where Test&Set and Fetch&Add2 are two atomic operations defined as follows:

- Test&Set returns the value of the memory location, and sets it to 1 if it contained 0,
- Fetch&Add2 returns the value in the memory location and increases it by 2.

Each of these operations on memory locations has consensus number 2. The algorithm described in Figure 7 (due to F. Ellen, G. Gelashvili, N. Shavit, and L. Zhu, [22]) shows that a binary consensus object can be built in the system model in $\mathcal{CARW}_n$[Test&Set, Fetch&Add2], for any value of $n$. This means that the previous hierarchy collapses when object types defined by operations on memory locations can be used to implement consensus. Binary consensus means that only the values 0 and 1 can be proposed. This is not a problem as it is possible to build a multivalued consensus object from binary consensus objects (see [52]).

```
when p_i invokes propose(v) do
(1)   if (v = 0) then X.fetch&add2();
(2)                   if (X is odd) then return(1) else return(0) end if
(3)             else  x ← X.test&set();
(4)                   if (x is odd) ∨ (x = 0) then return(1) else return(0) end if
(5)   end if.
```

Figure 7: A wait-free consensus algorithm in $\mathcal{CARW}_n$[Test&Set, Fetch&Add2] (code for process $p_i$)

The internal representation of the binary consensus object is a single memory location $X$, initialized to 0. According to the value it proposes (0 or 1), a process executes the statements of lines 2-3 or the statements of lines 4-5. The value returned by the consensus object is sealed by the first atomic operation that is executed. It is 0 if the first operation on $X$ is $X$.fetch&add2(), and 1 if first operation on $X$ is $X$.test&set(). The reader can check that, if the first operation on $X$ is fetch&add2(), $X$ becomes and remains even forever. If it is test&set(), $X$ becomes and remains odd forever. In the first case, only 0 can be decided, while in the second case, only 1 can be decided.

**Power number** The notion of the *power number* of an object type $T$ (PN($T$)) was introduced by G. Taubenfeld in [59]. It is the largest integer $k$ such that it is possible to implement a $k$-obstruction-free consensus object for *any* number of processes, using any number of atomic read/write registers, and any number of objects of type $T$ (the registers and the objects of type $T$ being wait-free). If there is no such largest $k$, PN($T$) = $+\infty$.

Hence, the power number of an object type $T$ relates $k$-obstruction-freedom and wait-freedom, when objects of type $T$ are used. Let CN($T$) be the consensus number of the objects of type $T$. It is shown in [59] that CN($T$) = PN($T$). This result establishes a strong relation linking wait-freedom and $k$-obstruction-freedom. As noticed in [59], "the difficult part of the proof is to show that, for any $k \geq 1$, it is possible to implement a $k$-obstruction-free consensus algorithm for any number of processes, using only wait-free consensus objects for $k$ processes and atomic read/write registers".

## 5.5 Universal construction "1 among $k$"

**$k$-Set agreement** $k$-Set agreement ($k$-SA) was introduced by S. Chaudhuri [19]. It is a simple generalization of consensus. It is defined by the same validity and termination properties, and a weaker agreement property, namely, at most $k$ different values can be decided by the processes. Hence, 1-set agreement is consensus.

It is shown in [10, 37, 56] that it is impossible to build a $k$-set agreement object in $\mathcal{CARW}_n[\emptyset]$ when $k$ or more processes may crash.

**$k$-simultaneous consensus** $k$-Simultaneous consensus ($k$-SC) was introduced in [3]. As consensus and $k$-SA, a $k$-SC object is a one-shot object that provides the processes with a single operation denoted propose(). This operation takes an input parameter a vector of size $k$, whose each entry contains a value, and returns a pair $\langle x, v\rangle$. The input vector contains "proposed" values, and if $\langle x, v\rangle$ is the pair returned to the invoking process, this process "decides $v$, and this decision is associated with the consensus instance $x$", $1 \leq x \leq k$.

More precisely, the behavior of a $k$-SC object is defined by the following properties.

- Termination. If a correct process invokes propose(), it decides a pair $\langle x, v\rangle$.
- Validity. If a process $p_i$ decides the pair $\langle x, v\rangle$, we have $1 \leq x \leq k$, and the value $v$ was proposed by a process in the entry $x$ of its input vector parameter.
- Agreement. Let $p_i$ be a process that decides the pair $\langle x, v\rangle$, and $p_j$ be a process that decides the pair $\langle y, w\rangle$. We have $(x = y) \Rightarrow (v = w)$.

It is shown in [3] that $k$-SA and $k$-SC have the same computational power in the sense that a $k$-SA object can be built in $\mathcal{CARW}_n[k\text{-SC}]$, and a $k$-SC object can be built in $\mathcal{CARW}_n[k\text{-SA}]$. This equivalence is no longer true in asynchronous crash-prone message-passing systems, where $k$-SC is stronger than $k$-SA [12, 54].

Let $in_i[1..k]$ be the input parameter of a process $p_i$. An easy implementation of $k$-SC in $\mathcal{CARW}_n[\emptyset]$ enriched with $k$ consensus objects $CONS[1..k]$ is as follows. For each $x$, $1 \leq x \leq k$, and in parallel, a process $p_i$ proposes $in_i[x]$ to the consensus object $CONS[x]$. Let $CONS[y]$ be the first consensus object which returns a value $v$ to $p_i$. Process $p_i$ decides then the pair $\langle y, v\rangle$.

**The notion of $k$-universality** E. Gafni and R. Guerraoui investigated in [27] the following question: What does happen if, instead of consensus objects, we use $k$-SA (or equivalently $k$-SC) objects to design a universal construction?

They showed that it is then possible to design what they called a *$k$-universal construction*. Such a construction considers $k$ objects (instead of only one) and guarantees that at least one of these objects progresses forever. Let GG denote the $k$-universal construction described in [27].

**Adopt-commit object** The GG construction relies on $k$-SC objects and adopt-commit (AC) objects. This object, introduced in [26], is a one-shot object which provides the processes with a single operation denoted propose(), which takes a value as input parameter and returns a pair composed of a tag and a value. Its behavior is defined by the following properties.

- Validity.
  - Result domain. Any returned pair $\langle tag, v\rangle$ is such that (a) $v$ has been proposed by a process and (b) $tag \in \{\texttt{commit}, \texttt{adopt}\}$.
  - No-conflicting values. If a process $p_i$ invokes propose($v$) and returns before any other process $p_j$ has invoked propose($w$) with $w \neq v$, only the pair $\langle \texttt{commit}, v\rangle$ can be returned.
- Agreement. If a process returns $\langle \texttt{commit}, v\rangle$, only the pairs $\langle \texttt{commit}, v\rangle$ or $\langle \texttt{adopt}, v\rangle$ can be returned by the other processes.
- Termination. An invocation of propose() by a correct process always terminates.

It follows from the "no-conflicting values" property that, if a single value $v$ is proposed, only the pair $\langle \texttt{commit}, v\rangle$ can be returned. Adopt-commit objects can be wait-free implemented in $\mathcal{CARW}_n[\emptyset]$ (e.g., [26, 52]). Hence, they provide processes with a higher abstraction level than read/write registers, but do not provide them with additional computational power.

**A non-blocking $k$-universal construction** (This section borrows text from [55]) The algorithm GG is based on local replication paradigm, namely, the only shared objects are the control objects $KSC[1..]$ (unbounded list of $k$-SC objects) and $AC[1..][1..k]$ (matrix of adopt-commit objects). Each process $p_i$ manages a copy of every object $m$, denoted $state_i[m]$, which contains the last state of $m$ as known by $p_i$. The invocation by $p_i$ of $\delta(state_i[m], \mathsf{op})$ applies the operation op() to its local copy of object $m$. The construction consists in an infinite sequence of asynchronous rounds, locally denoted $r_i$ at process $p_i$.

Each process manages the following local data structures.

- For each object $m$, $my\_list_i[m]$ defines the list of operations that $p_i$ wants to apply to the object $m$. Moreover, $my\_list_i[m]$.first() sets the read head to point to the first element of this list and returns its value; $my\_list_i[m]$.current() returns the operation under the read head; finally, $my\_list_i[m]$.next() advances the read head before returning the operation pointed to by the read head.

- For each object $m$, $oper_i[m]$, $ac\_op_i[m]$ are local variables which contain operations that $p_i$ wants to apply object $m$ (this list can be defined dynamically according to the algorithm executed by $p_i$); $tag_i[m]$ is used to contain a tag returned by an adopt-commit object concerning the object $m$.

The algorithm is presented in Figure 8. A process $p_i$ first initializes its round number, and the local copy of each object. The array $oper_i[1..k]$ is such that $oper_i[m]$ contains the next operation that $p_i$ wants to apply to $m$. When this is done, it enters an infinite loop, which constitutes the core of the construction. To simplify the presentation, and without loss of generality, we consider that all object operations are different (this can be easily realized with sequence numbers and process identities). Moreover, we also do not consider the result returned by each operation.

```
r_i ← 0;
for each m ∈ {1, ..., k} do
      state_i[m] ← initial state of the object m; oper_i[m] ← my_list_i[m].first()
end for.

repeat forever
(1)   r_i ← r_i + 1;
(2)   ⟨obj, op⟩ ← KSC[r_i].propose(oper_i[1..k]);
(3)   (tag_i[obj], ac_op_i[obj]) ← AC[r_i][obj].propose(op);
(4)   for each m ∈ {1, ..., k} \ {obj} do
            (tag_i[m], ac_op_i[m]) ← AC[r_i][m].propose(oper_i[m]) end for;
(5)   for each m ∈ {1, ..., k} do
(6)         if (ac_op_i[m] is marked "to_be_executed_after" oper_i[m])
(7)           then state_i[m].δ(state_i[m], oper_i[m])
(8)         end if;
(9)         if (oper_i[m] is not marked "to_be_executed_after" ac_op_i[m])
(10)          then  if (tag_i[m] = adopt)
(11)                  then oper_i[m] ← ac_op_i[m]
(12)                  else  state_i[m] ← δ(state_i[m], ac_op_i[m]); % tag_i[m] = commit %
(13)                        if ac_op_i[m] = my_list_i[m].current()
(14)                          then oper_i[m] ← my_list_i[m].next()
(15)                          else  oper_i[m] ← my_list_i[m].current()
(16)                        end if;
(17)                        mark oper_i[m] "to_be_executed_after" ac_op_i[m]
(18)                end if
(19)        end if
(20)  end for
end repeat.
```

Figure 8: Non-blocking $k$-universal construction (code of $p_i$)

After it has increased its round number, a process $p_i$ invokes the $k$-simultaneous consensus object $KSC[r]$ to which it proposes the operation vector $oper_i[1..n]$,

and from which it obtains the pair denoted $\langle obj, op\rangle$; $op$ is an operation proposed by some process for the object $obj$ (line 2). Process $p_i$ then invokes the adopt-commit object $AC[r][obj]$ to which it proposes the operation $op$ output by $KSC[r]$ for the object $obj$ (line 3). Finally, for all the other objects $m \neq obj$, $p_i$ invokes the adopt-commit object $AC[r][m]$ to which it proposes $oper_i[m]$ (line 4). As already indicated, the tags and the operations defined by the vector of pairs output by the adopt-commit objects $AC[r][1..k]$ are saved in the vectors $tag_i[1..k]$;and $ac\_op_i[1..k]$, respectively. The aim of these lines, realized by the objects $KSC[r]$ and $AC[r][1..m]$is to implement a filtering mechanism such that (a) for each object, at most one operation can be be committed, and (b) there is at least one object for which an operation is committed at some process. This filtering mechanism is explained separately below.

After the execution lines 2-4, for $1 \leq m \leq k$, $\langle tag_i[m], ac\_op_i[m]\rangle$ contains the operation that $p_i$ has to consider for the object $m$. For each of them it does the following. First, if $ac\_op_i[m]$ is marked "to be executed after" $oper_i[m]$, $p_i$ applies $oper_i[m]$ to $state_i[m]$ (lines 6-8). Then, the predicate of line 9 ensures that no operation invocation is applied twice on the same object (this line is missing in [27]). If $tag_i[m]$ = `adopt`, $p_i$ adopts $ac\_op_i[m]$ as its next proposal for the object $m$ (lines 10-11). Otherwise, $tag_i[m]$ = `commit`. In this case $p_i$ first applies $ac\_op_i[m]$ to its local copy $state_i[m]$ (line 12). Then, if $ac\_op_i[m]$ was an operation it has issued, $p_i$ computes its next operation $oper_i[m]$ on the object $m$ (lines 13-16).

As explained in [27], the use of a naive strategy to update local copies of the objects, makes possible the following bad scenario. During a round $r$, a process $p_1$ executes an operation op1 on its copy of the object $m1$, while a process $p_2$ executes a operation op2 on a different object $m2$. Then, during round $r + 1$, $p_1$ executes a operation op3 on the object $m2$ without having executed first op2 on its copy of $m2$. This bad behavior is prevented from occurring by a combined used of adopt-commit objects and an appropriate marking mechanism. When a process $p_i$ applies an operation op() to its local copy of an object $m$, it has necessarily received the pair $\langle$`commit`, op()$\rangle$ from the adopt-commit object associated with the current round, and consequently the other processes have received $\langle$`commit`, op()$\rangle$ or $\langle$`adopt`, op()$\rangle$. The process $p_i$ attaches then to its next operation for the object $m$ (which is denoted $oper_i[m]$) the indication that $oper_i[m]$ has to be applied to $m$ after op() so that no process executes $oper_i[m]$ without having previously executed op(). Hence, to prevent the bad behavior previously described, a process $p_i$ attaches to $oper_i[m]$ (line 17) the fact that this operation cannot be applied to any copy of the object $m$ before the operation $ac\_op_i[m]$.

As already indicated, this $k$-universal construction ensures that at least one process progresses forever (non-blocking progress condition), and at least one object progresses forever.

**Why at least one object operation is committed at every round** It was claimed above that the "filtering mechanism" realized by lines 2-4 ensures that at least one operation is committed at every round. We prove here this claim. Figure 9 illustrates the associated reasoning.

After (at line 2) a process $p_{i1}$ obtained a pair $\langle obj1, op1\rangle$ from its invocation $KSC[r]$.propose($oper_i[1..k]$), it invokes $AC[r][obj1]$.propose($op1$) at line 3, and only then it invokes $AC[r][obj]$.propose($op1$) for each object $obj \neq obj1$ at line 4. If its invocation of $AC[r][obj1]$.propose($op1$) at line 3 returns $\langle$`commit`, $-\rangle$, the claim follows.

Hence, let us assume that the invocation of $AC[r][obj1]$.propose($op1$) by $p_{i1}$ returns $\langle$`adopt`, $-\rangle$. It follows from the "non-conflicting" property of the AC object $AC[r][obj1]$ that another process $p_{i2}$ has necessarily invoked the operation $AC[r][obj1]$.propose($op'$) with $op' \neq op1$; moreover this invocation by $p_{i2}$ was issued at line 4 (if both $p_{i1}$ and $p_{i2}$ had invoked $AC[r][obj1]$.propose() at line 3, due to agreement property of $AC[r][obj1]$, they would have obtained the same pair from this object at line 3, and consequently $p_{i2}$ could not have prevented $p_{i1}$ from obtaining $\langle$`commit`, $-\rangle$ from the AC object $AC[r][obj1]$ at line 3). If follows that $p_{i2}$ started line 4 before $p_{i1}$ terminated line 3. The invocation by $p_{i2}$ at line 3 of $AC[r][-]$ involved some object $obj2$ obtained by $p_{i2}$ at line 2, and we necessarily have $obj2 \neq obj1$).

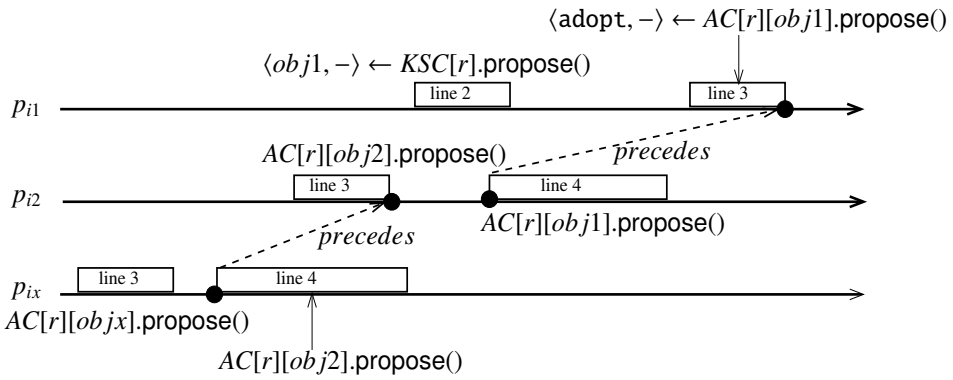



Figure 9: Net effect of the $k$-SC and CA objects used at lines 2-4 of round $r$

If the invocation of $AC[r][obj2]$.propose() returns $\langle$`commit`, $-\rangle$, the claim follows. Otherwise, due to the agreement property of $AC[r][obj2]$, there is a process $p_{i3}$, different from $p_{i1}$ and $p_{i2}$, such that the execution pattern between $p_{i3} \neq p_{i2}$ is the same as the previous pattern between $p_{i2} \neq p_{i1}$, etc. The claim then follows by induction and the fact that there is finite number of processes.

## 5.6 Ultimate universal construction "$\ell$ among $k$"

The previous NB-compliant $k$-universal construction ensures that at least one object progresses forever, and one process progresses forever. Hence, the natural question: Is it possible to design a universal construction in which at least $\ell$ objects progress forever, where $1 \leq \ell \leq k$, and all correct processes progress forever (wait-freedom progress condition).

Such a very general universal construction was proposed by M. Raynal, J. Stainer, and G. Taubenfeld in [55]. It rests on an extension of the $k$-SC object called $(k, \ell)$-simultaneous consensus.

**$(k, \ell)$-simultaneous consensus** Let $\ell \in \{1, ..., k\}$. A $(k, \ell)$-SC object is a $k$-SC object (see Section 5.5) where instead of a single pair $\langle x, v \rangle$, the operation propose() returns a set of exactly $\ell$ pairs $\{\langle x_1, v_1 \rangle, ..., \langle x_\ell, v_\ell \rangle\}$, such that all the pairs differ in their first component.

It is easy to see that $(k, 1)$-SC object is a $k$-SC object (and consequently a $k$-SA object). Moreover, a $(k, k)$-SC object is a consensus object. It is also easy to see that a $(k, k)$-SC object is a consensus object. For $k > 1$, a $(k, \ell)$-SC object is weaker than a $(k, \ell + 1)$-SC object.

**$(k, \ell)$-Universal construction** The $(k, \ell)$-universal construction presented in [55] borrows the lines 1-4 of Figure 8, in which $k$-SC objects are replaced by $(k, \ell)$-SC objects. All the rest of the construction, which is built incrementally, is based on a different approach. A non-blocking $k$-universal construction is first described, and then enriched step by step to obtain the final WF-compliant $(k, \ell)$-universal construction. Its noteworthy features are the following.

- On the object side. At least $\ell$ among the $k$ objects progress forever, $1 \leq \ell \leq k$. This means that an infinite number of operations is applied to each of these $\ell$ objects. This set of $\ell$ objects is not predetermined, and depends on the execution.
- On the process side. The progress condition associated with processes is wait-freedom. That is, a process that does not crash executes an infinite number of operations on each object that progresses forever.
- An object stops progressing when no more operations are applied to it. The construction guarantees that, when an object stops progressing, all its copies stop in the same state (at the non-crashed processes).
- The construction is *contention-aware*. This means that the overhead introduced by using operations on memory locations other than atomic read/write registers is eliminated when there is no contention during the execution of

an object operation. In the absence of contention, a process completes its operations by accessing only read/write registers.

- The construction is *generous* with respect to *obstruction-freedom*. This means that each process is able to complete its pending operations on all the $k$ objects each time all the other processes hold still long enough. That is, if once and again all the processes except one hold still long enough, then all the $k$ objects, and not just $\ell$ objects, are guaranteed to always progress.
- Last but least, it is shown in [55] that $(k, \ell)$-simultaneous consensus objects are necessary and sufficient to implement a $(k, \ell)$-universal construction, i.e. to ensure that at least $\ell$ among $k$ objects progress forever while guaranteeing the wait-freedom progress condition to the processes. Relations between $(k, k - p)$-SC objects and $(p + 1)$-set agreement objects for $0 \leq p < k$ are also investigated in [55].

# 6 Universal Construction vs Software Transactional Memory

A universal construction concerns the distributed implementation of concurrent objects defined by a sequential specification. The concept of a *software transactional memory* (STM), introduced in [35], and later refined in [57], is different. Its aim is to provide the programmers with a language construct (called *transaction*) that discharges them from the management of synchronization issues. In this way, a programmer can concentrate his efforts on which parts of processes have to be executed atomically and not on the way atomicity is realized. This last issue is then the job of the underlying STM system. Among others, main differences between universal constructions and STM systems are the following.

- Object operations are defined a priori (statically), and the universal construction knows them. Differently, the transactions are defined dynamically, and the STM system has no priori knowledge of their content and their effects.

  Let us also notice that, despite the fact they have the same name, database transactions [28] and STM transactions are not the same. Database transactions are constrained in the sense that they are the result of a queries expressed in a given formalism. Differently, STM transactions can be any piece of code produced by a programmer, which must be executed atomically. Moreover, usually the code of the STM transactions is not known by the STM system.
- The consistency condition of concurrent objects (captured at run-time by linearizability [38]) and the consistency conditions of STM systems (e.g.,

opacity [29], virtual world consistency [40], or TMS1 [20]) are different. Among other points, this come from the fact that any two transactions are a priori independent.

- Due to their very nature, universal constructions consider failure-prone systems. Differently, some STMs address failure-free systems while others address failure-prone systems.

# 7 Conclusion

The aim of this article was to be a guided visit to universal constructions in asynchronous crash-prone systems, where the processes communicate through a shared memory. As announced in the introduction, its ambition is not to be an exhaustive catalog of the numerous universal constructions proposed so far, but a relatively easy to understand introduction to the "universal construction" problem and the important concepts, objects, and approaches, which constitute the foundations of the associated algorithms.

To this end, the article has first presented a simple construction based on hardware operations on memory locations, namely the LL/SC pair of operations. It then moved from hardware-provided operations to agreement objects, and presented a simple consensus-based universal construction. Finally, the article considered the case where the aim is not to address the construction of a single object, but the coordinated construction of several objects. It is important to realize that, if not all the objects which are built are required to progress forever, hardware operations such as LL/SC or Compare&Swap are stronger than necessary to build universal constructions.

As a final remark, let us notice that OB-compliant (obstruction-free) universal constructions do not require to enrich the system with the additional computational power provided by instructions such as LL/SC or agreement objects, i.e., they can be done in the basic system model $\mathcal{CARW}[\emptyset]$. This remains true even if the processes are anonymous. The algorithms presented in [11] build a consensus object and a repeated consensus object respectively, in such an asynchronous crash-prone anonymous read/write system with only $n$ read/write atomic registers, which we conjecture to be optimal (it is proved in [63] that at least $(n-1)$ registers are necessary).

# Acknowledgments

This work was partially supported by the Franco-German DFG/ANR project DISCMAT devoted to connections between mathematics and distributed computing (ANR-14-CE35-0010-02), and the French ANR project DESCARTES devoted to distributed software engineering (ANR-16-CE40-0023-03). A special thank to Stefan Schmid for his careful reading of the article.


# References


[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

[2] Afek Y., Dauber D., and Touitou D., Wait-free made fast. *Proc. 27th ACM Symposium on Theory of Computing (STOC'95)*, ACM Press, pp. 538-547 (1995)

[3] Afek Y., Gafni E., Rajsbaum S., Raynal M., and Travers C., The $k$-simultaneous consensus problem. *Distributed Computing*, 22(3):185-195 (2010)

[4] Aguilera M.K., Frolund S., Hadzilacos V., Horn S.L., and Toueg S., Abortable and query-abortable objects and their efficient implementation. *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, ACM Press, pp. 23-32 (2007)

[5] Anderson J.H., Multi-writer composite registers. *Distributed Computing*, 7(4):175-195 (1994)

[6] Anderson J. and Moir M., Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317-1332 (1999)

[7] Attiya H., Bar-Noy A., Dolev D., Peleg D., and Reischuk R., Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524-548 (1990)

[8] Ben-David N., Cheng Chan D.Y., Hadzilacos V. and Toueg S., k-Abortable objects: progress under high contention. *Proc. 30th Int'l Symposium on Distributed Computing (DISC'16)*, Springer LNCS 9888, pp. 298-312 (2016)

[9] Bartlett K. A., Scantlebury S. A., and Wilkinson P. T., A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260-261 (1969)

[10] Borowsky E. and Gafni E., Generalized FLP impossibility results for $t$-resilient asynchronous computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100 (1993)

[11] Bouzid Z., Raynal M., and Sutra P., Anonymous obstruction-free $(n, k)$-set agreement with $(n - k + 1)$ atomic read/write registers. *Proc. 19th Int'l Conference On Principles Of Distributed Systems (OPODIS'15)*, Leibniz Int'l Proceedings in Informatics, LIPICS 46, Article 18:1-17 (2015)

[12] Bouzid Z. and Travers C., Simultaneous consensus is harder than set agreement in message-passing. *Proc. 33rd Int'l IEEE Conference on Distributed Computing Systems (ICDCS'13)*, IEEE Press, pp. 611-620 (2013)

[13] Brinch Hansen, P., *The origin of concurrent programming*. Springer, 534 pages, ISBN 0-387-95401-5 (2002)

[14] Bushkov V. and Guerraoui G., Safety-liveness exclusion in distributed computing. *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 227-236 (2015)

[15] Capdevielle Cl., Johnen C., and Milani A., Solo-fast universal constructions for deterministic abortable objects. *Proc. 28th Int'l Symposium on Distributed Computing (DISC'14)*, Springer LNCS 8784, pp. 288-302 (2014)

[16] Castañeda A., Rajsbaum S., and Raynal M., The renaming problem in shared memory systems: an introduction. *Elsevier Computer Science Review*, 5:229-251 (2011)

[17] Censor-Hillel K., Petrank E., and Timnat S., Help! *Proc. 34th Symposium on Principles of Distributed Computing (PODC'15)*, ACM Press, pp. 241-250 (2015)

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

[19] Chaudhuri S., More choices allow more faults: set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132-158 (1993)

[20] Doherty S., Groves L., Luchangco V., and Moir M., Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25:769-799 (2013)

[21] Ellen F., Fatourou P., Kosmas E., Milani A., and Travers C., Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distributed Computing*, 29:251-277 (2016)

[22] Ellen F., Gelashvili G., Shavit N. and Zhu L., A complexity-based hierarchy for multiprocessor synchronization (Extended abstract). *Proc. 35th ACM Symposium on Principles of Distributed Computing (PODC'16)*, ACM Press, pp. 289-298 (2016)

[23] Fatourou P. and Kallimanis N.D., The RedBlue adaptive universal constructions. *Proc. 23rd Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 127-141 (2009)

[24] Fatourou P. and Kallimanis N.D., Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55:475-520 (2014)

[25] Fischer M.J., Lynch N.A., and Paterson M.S., Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374-382 (1985)

[26] Gafni E., Round-by-round fault detectors: unifying synchrony and asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC)*, ACM Press, pp. 143-152 (1998)

[27] Gafni E. and Guerraoui R., Generalizing universality. *Proc. 22nd Int'l Conference on Concurrency Theory (CONCUR'11)*, Springer LNCS 6901, pp. 17-27 (2011)

[28] Gray J., Notes on database operating systems. *Advanced course on Operating Systems*, Springer LNCS 60, pp. 393-481 (1978)

[29] Guerraoui R. and Kapalka M., On the correctness of transactional memory. *Proc. 3rd ACM Symposium on Principles an Practice of Parallel Programming (PPOPP'03)*, ACM Press, pp. 175-184 (2008)

[30] Guerraoui R. and Raynal M., A universal construction for wait-free objects. *Proc. Workshop on Foundations of Fault-Tolerant Distributed Computing (FOFDC 2007)*, Computer Society Press, pp. 959-966 (2007)

[31] Hadzilacos V. and Toueg S., On deterministic abortable objects. *Proc. 35th ACM symposium on Principles of Distributed Computing (PODC'13)*, ACM Press, pp. 4-12 (2013)

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

[33] Herlihy M.P., A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745-770 (1993)

[34] Herlihy M.P., Luchangco V., and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, IEEE Press, pp. 522-529 (2003)

[35] Herlihy M. and Moss J.E.B., Transactional memory: architectural support for lock-free data structures. *Proc. 20th Annual International Symposium on Computer Architecture (ISCA'93)*, ACM Press, pp. 289-300 (1993)

[36] Herlihy M., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509:3-24 (2013)

[37] Herlihy M.P. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923 (1999)

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

[39] Imbs D. and Raynal M., Help when needed, but no more: efficient read/write partial snapshot. *Journal of Parallel and Distributed Computing*, 72(1):1-13 (2012)

[40] Imbs D. and Raynal M., Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113-127 (2012)

[41] Imbs D., Raynal M., and Taubenfeld G., On asymmetric progress conditions. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 55-64 (2010)

[42] Kramer S. N., *History begins at Sumer: thirty-nine firsts in man's recorded history*. University of Pennsylvania Press, 416 pages, ISBN 978-0-8122-1276-1 (1956)

[43] Lamport L., Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565 (1978)

[44] Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

[45] Lamport L., Fast mutual exclusion. *ACM Transactions on Computer Systems*, 5(1):1-11 (1987)

[46] Lamport L., Shostack R. and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

[47] Loui M. and Abu-Amara H., Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163-183, JAI Press (1987)

[48] Lynch W. C., Reliable full-duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407-410 (1968)

[49] Mostéfaoui A., Perrin M., and Raynal M., A simple object that spans the whole consensus hierarchy. *Submitted to publication*, (2016)

[50] Neugebauer O. E., *The exact sciences in Antiquity.* Princeton University Press (1952); 2nd edition: Brown University Press, (1957); Reprint: Dover publications (1969)

[51] Post E. L., Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65 (2):197-215 (1943)

[52] Raynal M., *Concurrent programming: algorithms, principles and foundations*. Springer, 515 pages, ISBN 978-3-642-32026-2 (2013)

[53] Raynal M., Concurrent systems: hybrid object implementations and abortable objects. *Proc. 21th Int'l European Parallel Computing Conference (EUROPAR'15)*, Springer LNCS 9233, pp. 3-15 (2015)

[54] Raynal M. and Stainer J., Simultaneous consensus vs set agreement: a message-passing-sensitive hierarchy of agreement problems. *Proc. 20th Int'l Colloquium on Structural Information and Communication Complexity (SIROCCO 2013)*, Springer LNCS 8179, pp. 298-309 (2013)

[55] Raynal M., Stainer J., and Taubenfeld G., Distributed universality. *Algorithmica*, 76(2):502-535 (2016)

[56] Saks M. and Zaharoglou F., Wait-free $k$-set agreement is impossible: the topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449-1483 (2000)

[57] Shavit N. and Touitou D., Software transactional memory. *Distributed Computing* 10(2):99-116 (1997)

[58] Taubenfeld G., *Synchronization algorithms and concurrent programming*. 423 pages, Pearson Education/Prentice Hall, ISBN 0-131-97259-6 (2006)

[59] Taubenfeld G., Contention-sensitive data structure and algorithms. *Proc. 23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer LNCS 5805, pp. 157-171 (2009)

[60] Taubenfeld G., The computational structure of progress conditions. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer LNCS 6343, pp. 221-235 (2010)

[61] Turing A. M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265 (1936)

[62] Wantzel P. L., Recherches sur les moyens de reconnaître si un problème de géométrie peut se résoudre avec la règle et le compas, *Journal de mathématiques pures et appliquées*, 1(2):366-372 (1837)

[63] Zhu L., A tight space bound for consensus. *Proc. 48th ACM Symposium on Theory of Computing (STOC'16)*, ACM Press, pp. 345-350 (2016)

# LOWER BOUNDS FOR TRANSACTIONAL MEMORY

Srivatsan Ravi
Purdue University

**Abstract**

Transactional memory allows the user to declare sequences of instructions as speculative *transactions* that can either *commit* or *abort*. If a transaction commits, it appears to be executed sequentially, so that the committed transactions constitute a correct sequential execution. If a transaction aborts, none of its update operations can affect other transactions. The TM implementation endeavors to execute these instructions in a manner that efficiently utilizes the concurrent computing facilities provided by multicore architectures.

The TM abstraction, in its original manifestation, extended the processor's instruction set with instructions to indicate which memory accesses must be transactional. Most popular TM designs, subsequent to the original proposal have implemented all the functionality in *software*. More recently, processors have included hardware extensions to support small transactions. Hardware transactions may be spuriously aborted due to several reasons: cache capacity overflow, interrupts *etc*. This has led to proposals for *hybrid* TMs in which the fast, but potentially unreliable hardware transactions are complemented with slower, but more reliable software transactions.

The complexity of TM implementations, whether realized in hardware or software, is characterized by several measures: ordering semantics for transactions, conditions under which transactions must terminate, conditions under which transactions must commit/abort, bound on the number of versions that can be maintained, choice of the complexity metric and complexity of *read-only* or *updating* transactions as well as a multitude of other implementation strategies. In this work, we survey known complexity bounds for implementing TM as a shared object and the implicit assumptions underlying these results.

# 1 Introduction

*The wickedness and the foolishness of no man can avail against the fond optimism of mankind.*
*James Branch Cabell-The Silver Stallion*

Transactional memory (TM) allows concurrent processes to organize sequences of operations on shared *data items* into atomic transactions. A transaction may commit, in which case its updates of data items "take effect" or it may *abort*, in which case no data items are updated. A TM *implementation* provides processes with algorithms for implementing transactional operations on data items (such as *read*, *write* and *tryCommit*) by applying *primitives* on shared *base objects*. Intuitively, the idea behind the TM abstraction is *optimism*: before a transaction commits, all its operations are speculative, and it is expected that, in the absence of concurrency, a transaction commits.

TM implementations typically ensure that all committed transactions appear to execute sequentially in some total order respecting the timing of non-overlapping transactions. Moreover, intermediate states witnessed by the read operations of an incomplete transaction may affect the user application. Thus, to ensure that the TM implementation is *safe* and does not export any pathological executions, it is additionally expected that every transaction (including aborted and incomplete ones) must return responses that is consistent with some *sequential* execution of the TM implementation.

**TM implementations.** As a *synchronization abstraction*, TM came as an alternative to conventional lock-based synchronization. The TM abstraction, in its original manifestation, augmented the processor's *cache-coherence protocol* and extended the CPU's instruction set with instructions to indicate which memory accesses must be transactional [39]. Most popular TM designs, subsequent to the original proposal in [39] have implemented all the functionality in software [18, 29, 38, 50, 61]. Early software transactional memory (STM) implementations [29, 38, 50, 61, 63] adopted optimistic concurrency control and guaranteed that a prematurely halted transaction cannot not prevent other transactions from committing. These implementations avoided using locks and relied on *non-blocking* (sometimes also called *lock-free*) synchronization. Possibly the weakest non-blocking progress condition is *obstruction-freedom* [37, 40] stipulating that every transaction running in the absence of *step contention*, *i.e.*, not encountering steps of concurrent transactions, must commit.

In 2005, Ennals [28] argued that that obstruction-free TMs inherently yield poor performance, because they require transactions to forcefully abort each other. Ennals further describes a *lock-based* TM implementation [27] that he claimed to outperform *DSTM* [38], the most referenced obstruction-free TM implementation at

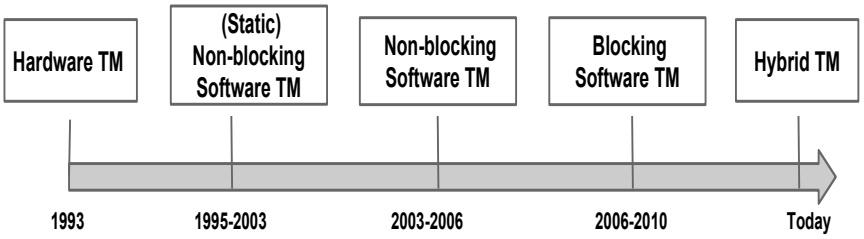



Figure 1: TM implementations: a brief history

the time. Inspired by [28], more recent TM implementations like *TL* [21], *TL2* [20] and *NOrec* [18] employ locking and showed that Ennal's claims about performance of lock-based TMs hold true on most workloads. The progress guarantee provided by these TMs is typically *progressiveness*: a transaction may be aborted only if it encounters a read-write or a write-write conflicts with a concurrent transaction [32]. Nonetheless, TM designs that are implemented entirely in software still incur significant performance overhead. Thus, current CPUs have included instructions to mark a block of memory accesses as transactional [1, 53, 56], allowing them to be executed *atomically* in hardware. Hardware transactions promise better performance than STMs, but they offer no progress guarantees since they may experience *spurious* aborts. This motivates the need for *hybrid* TMs in which the *fast* hardware transactions are complemented with *slower* software transactions that do not have spurious aborts.

**Our focus.** This work surveys lower bounds and (im)possibility results for TM implementations. We identify the popular complexity metrics (e.g. *expensive synchronization patterns* [8], *memory stalls* [26], number of memory steps etc.) and their relevance in the TM context. We survey known lower and upper bounds on the complexity of three classes of safe (software) TMs: *blocking* TMs that allow transactions to delay or abort due to *overlapping* transactions (Section 3), *non-blocking* TMs which adapt to *step contention* by ensuring that a transaction not encountering steps of overlapping transactions must commit (Section 4), and *partially non-blocking* TMs that provide strong non-blocking guarantees (*wait-freedom*) to only a subset of transactions (Section 5). We then survey attempts at modelling HyTMs and lower bounds that exhibit inherent trade-offs on the degree of concurrency allowed between hardware and software transactions and the costs introduced on the hardware (Section 6). We conclude with an overview of future research directions and open questions concerning complexity of TMs (Section 7).

## 2 Transactional memory model and preliminaries

**TM interface.** *Transactional memory* (in short, *TM*) allows a set of data items (called *t-objects*) to be accessed via atomic *transactions*. A transaction $T_k$ may contain the following *t-operations*: $read_k(X)$ returns a value in some domain $V$ (denoted $read_k(X) \to v$) or a special value $A_k \notin V$ (*abort*); $write_k(X, v)$, for a value $v \in V$, returns *ok* or $A_k$; $tryC_k$ returns $C_k \notin V$ (*commit*) or $A_k$.

**TM implementations.** We consider an asynchronous shared-memory system in which a set of $n$ processes, communicate by applying *primitives* on shared *base objects*. We assume that processes issue transactions sequentially, *i.e.*, a process starts a new transaction only after its previous transaction has *completed* (committed or aborted). A TM *implementation* provides processes with algorithms for implementing $read_k$, $write_k$ and $tryC_k()$ of a transaction $T_k$ by *applying primitives* from a set of shared *base objects*, each of which is assigned an *initial value*. A primitive is a generic *read-modify-write* (*rmw*) procedure applied to a base object [26, 36]. It is characterized by a pair of functions $\langle g, h\rangle$: given the current state of the base object, $g$ is an *update function* that computes its state after the primitive is applied, while $h$ is a *response function* that specifies the outcome of the primitive returned to the process. A rmw primitive is *trivial* if it never changes the value of the base object to which it is applied. Otherwise, it is *nontrivial*. A trivial rmw primitive is *conditional* if there exist configurations in which the primitive does not change the value of the base object. Observe that this model explicitly precludes the use of atomic primitives that access multiple base objects in a single step [24].

**Executions and configurations.** An *event* of a transaction $T_k$ (sometimes we say a *step* of $T_k$) is a rmw primitive $\langle g, h\rangle$ applied by $T_k$ to a base object $b$ along with its response $r$ (we call it a *rmw event* and write $(b, \langle g, h\rangle, r, k)$), or the invocation or the response of a t-operation performed by $T_k$.

A *configuration* (of a TM implementation) specifies the value of each base object and the state of each process. The *initial configuration* is the configuration in which all base objects have their initial values and all processes are in their initial states.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* of a TM implementation $M$ is an execution fragment where, starting from the initial configuration, each event is issued according to $M$ and each response of a RMW event $(b, \langle g, h\rangle, r, k)$ matches the state of $b$ resulting from the preceding events. If an execution can be represented as $E \cdot E'$ (concatenation of execution fragments $E$ and $E'$), then we say that $E \cdot E'$ is an *extension* of $E$ or $E'$ *extends* $E$.

Let $E$ be an execution fragment. For a transaction $T_k$ (and resp. process $p_k$), $E|k$ denotes the subsequence of $E$ restricted to events of $T_k$ (and resp. $p_k$). If $E|k$ is non-empty, we say that $T_k$ (resp. $p_k$) *participates* in $E$, else we say $E$ is

$T_k$-*free* (resp. $p_k$-*free*). Two executions $E$ and $E'$ are *indistinguishable* to a set $\mathcal{T}$ of transactions, if for each transaction $T_k \in \mathcal{T}$, $E|k = E'|k$. A TM *history* is the subsequence of an execution consisting of the invocation and response events of t-operations. Two histories $H$ and $H'$ are *equivalent* if $txns(H) = txns(H')$ and for every transaction $T_k \in txns(H)$, $H|k = H'|k$.

**Dynamic programming model.** The *read set* (resp., the *write set*) of a transaction $T_k$ in an execution $E$, denoted $Rset_E(T_k)$ (and resp. $Wset_E(T_k)$), is the set of t-objects that $T_k$ attempts to read (and resp. write) by issuing a t-read (and resp. t-write) invocation in $E$ (for brevity, we sometimes omit the subscript $E$ from the notation). The *data set* of $T_k$ is $Dset(T_k) = Rset(T_k) \cup Wset(T_k)$. $T_k$ is called *read-only* if $Wset(T_k) = \emptyset$; *write-only* if $Rset(T_k) = \emptyset$ and *updating* if $Wset(T_k) \neq \emptyset$. Note that we consider the conventional *dynamic TM model*: the data set of a transaction is identifiable only by the set of t-objects the transaction has invoked a read or write in the given execution.

**Orders on transactions.** Let $txns(E)$ denote the set of transactions that participate in $E$. An execution $E$ is *sequential* if every invocation of a t-operation is either the last event in the history $H$ exported by $E$ or is immediately followed by a matching response. We assume that executions are *well-formed*, *i.e.*, for all $T_k$, $E|k$ begins with the invocation of a t-operation, is sequential and has no events after $A_k$ or $C_k$. A transaction $T_k \in txns(E)$ is *complete in* $E$ if $E|k$ ends with a response event. The execution $E$ is *complete* if all transactions in $txns(E)$ are complete in $E$. A transaction $T_k \in txns(E)$ is *t-complete* if $E|k$ ends with $A_k$ or $C_k$; otherwise, $T_k$ is *t-incomplete*. $T_k$ is *committed* (resp., *aborted*) in $E$ if the last event of $T_k$ is $C_k$ (resp., $A_k$). The execution $E$ is *t-complete* if all transactions in $txns(E)$ are t-complete.

For transactions $\{T_k, T_m\} \in txns(E)$, we say that $T_k$ *precedes* $T_m$ in the *real-time order* of $E$, denoted $T_k \prec^{RT}_E T_m$, if $T_k$ is t-complete in $E$ and the last event of $T_k$ precedes the first event of $T_m$ in $E$. If neither $T_k \prec^{RT}_E T_m$ nor $T_m \prec^{RT}_E T_k$, then $T_k$ and $T_m$ are *concurrent* in $E$. An execution $E$ is *t-sequential* if there are no concurrent transactions in $E$.

**Contention.** We say that a configuration $C$ after an execution $E$ is *quiescent* (resp., *t-quiescent*) if every transaction $T_k \in txns(E)$ is complete (resp., t-complete) in $C$. If a transaction $T$ is incomplete in an execution $E$, it has exactly one *enabled* event, which is the next event the transaction will perform according to the TM implementation. Events $e$ and $e'$ of an execution $E$ *contend* on a base object $b$ if they are both events on $b$ in $E$ and at least one of them is nontrivial (the event is trivial (resp., nontrivial) if it is the application of a trivial (resp., nontrivial) primitive).

We say that $T$ is *poised to apply an event* $e$ *after* $E$ if $e$ is the next enabled event for $T$ in $E$. We say that transactions $T$ and $T'$ *concurrently contend on* $b$ *in* $E$ if

they are poised to apply contending events on $b$ after $E$.

We say that an execution fragment $E$ is *step contention-free for t-operation* $op_k$ if the events of $E|op_k$ are contiguous in $E$. We say that an execution fragment $E$ is *step contention-free for* $T_k$ if the events of $E|k$ are contiguous in $E$. We say that $E$ is *step contention-free* if $E$ is step contention-free for all transactions that participate in $E$.

**TM-correctness.** Informally, a t-sequential history $S$ is *legal* if every t-read of a t-object returns the *latest written value* of this t-object. A history $H$ is *opaque* if there exists a legal t-sequential history $S$ equivalent to $H$ such that $S$ respects the real-time order of transactions in $H$ [33].

A weaker condition called *strict serializability* ensures opacity only with respect to committed transactions while definitions like virtual-world consistency (VWC) [41] and transactional memory specification (TMS1) ensure strict serializability and restricted safety for aborted transactions [25]. We direct the reader to [9] for details on these definitions.

**TM-progress.** One may notice that a TM implementation that forces, in every execution to abort every transaction is trivially strictly serializable, but not very useful. A TM-progress condition specifies the conditions under which a transaction is allowed to abort. Technically, a TM-progress condition specified this way is a *safety property* since it can be violated in a finite execution.

**TM-liveness.** Observe that a TM-progress condition only specifies the conditions under which a transaction is aborted, but does not specify the conditions under which it must commit. Thus, in addition to a progress condition, we must stipulate a *liveness* [5, 49] condition.

**Read invisibility.** Informally, in a TM using *invisible reads*, a transaction cannot reveal any information about its read set to other transactions. Thus, given an execution $E$ and some transaction $T_k$ with a non-empty read set, transactions other than $T_k$ cannot distinguish $E$ from an execution in which $T_k$'s read set is empty. This prevents TMs from applying nontrivial primitives during t-read operations and from announcing read sets of transactions during tryCommit. Most popular TM implementations like *TL2* [20] and *NOrec* [18] satisfy this property.

The notion of *weak* invisible that prevents t-read operations from applying nontrivial primitives only in the absence of concurrent transactions. Specifically, weak read invisibility allows t-read operations of a transaction $T$ to be "visible", *i.e.*, write to base objects, only if $T$ is concurrent with another transaction. For example, the popular TM implementation *DSTM* [38] satisfies weak invisible reads, but not invisible reads.

**Disjoint-access parallelism (DAP).** A TM implementation $M$ is *strictly disjoint-access parallel (strict DAP)* if, for all executions $E$ of $M$, and for all transactions $T_i$ and $T_j$ that participate in $E$, $T_i$ and $T_j$ contend on a base object in $E$ only if

$Dset(T_i) \cap Dset(T_j) \neq \emptyset$ [33].

Informally, *weak* DAP [11] ensures that two transactions concurrently contend on the same base object only if their data sets are connected in the *conflict graph*, capturing data-set overlaps among all concurrent transactions [11]. *Read-write (RW)* DAP [45], a restriction of weak DAP and a relaxation of strict DAP, defines the conflict graph based on the *write-set overlaps* among concurrent transactions and is satisfied by several popular obstruction-free implementations [29, 38, 63].

Observe that every RW DAP TM implementation satisfies weak DAP, but not vice versa. Consider the following execution $E$ that begins with the incomplete execution of a transaction $T_0$ that reads $X$ and writes to $Y$, followed by the execution of two transactions $T_1$ and $T_2$ that access $X$ and $Y$ respectively. If $E$ is an execution of a weak DAP TM, transactions $T_1$ and $T_2$ may contend on a base object since there is a path between $X$ and $Y$ in $G(T_1, T_2, E)$. However, a RW DAP TM implementation would preclude transactions $T_1$ and $T_2$ from contending on the same base object: there is no edge between t-objects $X$ and $Y$ in the corresponding conflict graph $\tilde{G}(T_1, T_2, E)$ because $X$ and $Y$ are not contained in the write set of $T_0$.

For any two DAP definitions $D_1$ and $D_2$, if every TM implementation that satisfies $D_1$ also satisfies $D_2$, but the converse is not true, we say that $D_2 \ll D_1$. The following observation is immediate.

**Observation 1.** *Weak DAP* $\ll$ *RW DAP* $\ll$ *Strict DAP* $\ll$ *Strict data-partitioning.*

# 3 Complexity of blocking TMs

We begin by overviewing TM implementations that are *blocking*. Figure 2 characterizes the class of blocking TMs: *single-lock* TMs that satisfy *sequential* TM-progress (a transaction may abort due to a concurrent transaction), *(strongly) progressive* TMs that allow transactions to abort only due to read-write conflicts on t-objects and finally *permissive* TMs that provide maximal concurrency allowing a transaction to abort only if committing it would violate TM-correctness.

## 3.1 Sequential TMs

**A quadratic lower bound on step complexity.** [47] showed that a read-only transaction in an opaque TM featured with weak DAP, weak invisible reads, *interval contention-free* (ICF) TM-liveness and sequential TM-progress must *incrementally* validate every next read operation. This results in a quadratic (in the size of the transaction's read set) step-complexity lower bound. Here ICF TM-liveness means, for every finite execution $E$ such that the configuration after $E$ is quiescent,

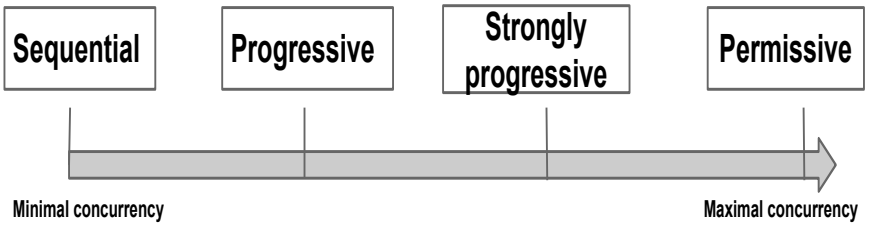



Figure 2: Classification of blocking TMs based on TM-progress: sequential (aborts due to concurrent transaction); progressive (aborts due to read-write conflicts; strongly progressive (progressive but at least one transaction from a set conflicting on single t-object must not be aborted); permissive (abort only due to TM-correctness violation)

and every transaction $T_k$ that applies the invocation of a t-operation $op_k$ immediately after $E$, the finite step contention-free extension for $op_k$ contains a response. Secondly, [47] prove that if the TM-correctness property is weakened to strict serializability, there exist executions in which the tryCommit of some transaction must access a linear (in the size of the transaction's read set) number of distinct base objects.

**Theorem 2** ( [47])**.** *For every weak DAP TM implementation $M$ that provides ICF TM-liveness, sequential TM-progress and uses weak invisible reads,*

*(1) If $M$ is opaque, for every $m \in \mathbb{N}$, there exists an execution $E$ of $M$ such that some transaction $T \in txns(E)$ performs $\Omega(m^2)$ steps, where $m = |Rset(T_k)|$.*

*(2) if $M$ is strictly serializable, for every $m \in \mathbb{N}$, there exists an execution $E$ of $M$ such that some transaction $T_k \in txns(E)$ accesses at least $m - 1$ distinct base objects during the executions of the $m^{th}$ t-read operation and $tryC_k()$, where $m = |Rset(T_k)|$.*

Theorem 2 improves the read-validation step-complexity lower bound [32, 33] derived for *strict-data partitioning* (a very strong version of DAP) and *invisible reads*. In a *strict data partitioned* TM, the set of base objects used by the TM is split into disjoint sets, each storing information only about a single data item. Indeed, every TM implementation that is strict data-partitioned satisfies weak DAP, but not vice-versa. The definition of invisible reads assumed in [32,33] requires that a t-read operation does not apply nontrivial events in any execution. Theorem 2 however, assumes *weak* invisible reads, stipulating that t-read operations of a transaction $T$ do not apply nontrivial events only when $T$ is not concurrent with any other transaction. We believe that the TM-progress and TM-liveness restrictions as well as the definitions of DAP and invisible reads considered for this result are

| TM-correctness | TM-liveness | DAP | Invisible reads | Read-write | Complexity |
|---|---|---|---|---|---|
| Opacity | ICF | weak | yes | yes | $\Theta(\|Rset\|^2)$ step-complexity |
| Strict serializability | ICF | weak | yes | yes | $\Theta(\|Rset\|)$ step-complexity for tryCommit |
| Opacity | WF | strict | yes | yes | $O(1)$ RAW/AWAR, $O(1)$ stalls for t-reads |
| Opacity | starvation-free | strict | | | $\Theta(\|Wset\|)$ protected data |

Table 1: Complexity bounds for progressive TMs.

the weakest possible assumptions that may be made. To the best of our knowledge, these assumptions cover every TM implementation that is subject to the validation step-complexity [18, 20, 38].

## 3.2 Progressive TMs

We turn our focus to *progressive* TM implementations which allow a transaction to be aborted only due to read-write conflicts with concurrent transactions.

**A linear lower bound on protected data size.** Kuznetsov *et al.* [44] introduce a new metric called *protected data size* that, intuitively, captures the amount of data that a transaction must exclusively control at some point of its execution. All progressive TM implementations (see, *e.g.*, an overview in [32]) use locks or timing assumptions to give an updating transaction exclusive access to all objects in its write set at some point of its execution. For example, lock-based progressive implementations like *TL* [21] and *TL2* [20] require that a transaction grabs all locks on its write set before updating the corresponding base objects. [44] shows that this is an inherent price to pay for providing progressive concurrency: every committed transaction in a progressive and strict DAP TM implementation providing *starvation-free* (each t-operation eventually returns a matching response, assuming that no concurrent t-operation stops indefinitely before returning) TM-liveness must, at some point of its execution, protect every t-object in its write set.

Let $M$ be a progressive TM implementation providing starvation-free TM-liveness. Intuitively, a t-object $X_j$ is protected at the end of some finite execution $\pi$ of $M$ if some transaction $T_0$ is about to atomically change the value of $X_j$ in its next step (*e.g.*, by performing a compare-and-swap) or does not allow any concurrent transaction to read $X_j$ (*e.g.*, by holding a "lock" on $X_j$).

Formally, let $\alpha \cdot \pi$ be an execution of $M$ such that $\pi$ is a t-sequential t-complete execution of a transaction $T_0$, where $Wset(T_0) = \{X_1, \ldots, X_m\}$. Let $u_j$ ($j = 1, \ldots, m$) denote the value written by $T_0$ to t-object $X_j$ in $\pi$. In this section, let $\pi^t$ denote the $t$-th shortest prefix of $\pi$. Let $\pi^0$ denote the empty prefix.

For any $X_j \in Wset(T_0)$, let $T_j$ denote a transaction that tries to read $X_j$ and commit. Let $E^t_j = \alpha \cdot \pi^t \cdot \rho^t_j$ denote the extension of $\alpha \cdot \pi^t$ in which $T_j$ runs solo until it completes. Note that, since we only require the implementation to be starvation-free, $\rho^t_j$ can be infinite.

We say that $\alpha \cdot \pi^t$ is $(1, j)$-valent if the read operation performed by $T_j$ in $\alpha \cdot \pi^t \cdot \rho^t_j$ returns $u_j$ (the value written by $T_0$ to $X_j$). We say that $\alpha \cdot \pi^t$ is $(0, j)$-valent if the read operation performed by $T_j$ in $\alpha \cdot \pi^t \cdot \rho^t_j$ does not abort and returns an "old" value $u \neq u_j$. Otherwise, if the read operation of $T_j$ aborts or never returns in $\alpha \cdot \pi^t \cdot \rho^t_j$, we say that $\alpha \cdot \pi^t$ is $(\bot, j)$-valent.

**Definition 1** ( [44])**.** *We say that* $T_0$ protects *an object $X_j$ in $\alpha \cdot \pi^t$, where $\pi^t$ is the $t$-th shortest prefix of $\pi$ ($t > 0$) if one of the following conditions holds: (1) $\alpha \cdot \pi^t$ is $(0, j)$-valent and $\alpha \cdot \pi^{t+1}$ is $(1, j)$-valent, or (2) $\alpha \cdot \pi^t$ or $\alpha \cdot \pi^{t+1}$ is $(\bot, j)$-valent.*

**Theorem 3** ( [44])**.** *Let $M$ be a progressive, opaque and strict disjoint-access-parallel TM implementation that provides starvation-free TM-liveness. Let $\alpha \cdot \pi$ be an execution of $M$, where $\pi$ is a t-sequential t-complete execution of a transaction $T_0$. Then, there exists $\pi^t$, a prefix of $\pi$, such that $T_0$ protects $|Wset(T_0)|$ t-objects in $\alpha \cdot \pi^t$.*

**A constant stall and constant expensive synchronization strict DAP opaque TM.** Attiya *et al.* identified two common expensive synchronization patterns that frequently arise in the design of concurrent algorithms: *read-after-write (RAW) or atomic write-after-read (AWAR)* [8, 52] and showed that it is impossible to derive RAW/AWAR-free implementations of a wide class of data types that include *sets*, *queues* and *deadlock-free mutual exclusion*. RAW (read-after-write) or AWAR (atomic-write-after-read) patterns [8] capture the amount of "expensive synchronization", *i.e.*, the number of costly memory barriers or conditional primitives [2] incurred by the implementation in relaxed CPU architectures. The metric appears to be more practically relevant than simply counting the number of steps performed by a process, as it accounts for expensive cache-coherence operations or instructions like compare-and-swap.

A *RAW* (read-after-write) pattern performed by a transaction $T_k$ in an execution $\pi$ is a pair of its events $e$ and $e'$, such that: (1) $e$ is a write to a base object $b$ by $T_k$, (2) $e'$ is a subsequent read of a base object $b' \neq b$ by $T_k$, and (3) no event on $b$ by $T_k$ takes place between $e$ and $e'$. Note that we are concerned only with *non-overlapping* RAWs, *i.e.*, the read performed by one RAW precedes the write performed by the other RAW. An *AWAR* (atomic-write-after-read) pattern $e$ in an execution $\pi \cdot e$ is a nontrivial rmw event on an object $b$ which atomically returns the value of $b$ (resulting after $\pi$) and updates $b$ with a new value.

Intuitively, the stall metric captures the fact that the time a process might have to spend before it applies a primitive on a base object can be proportional to the

number of processes that try to update the object concurrently. Let $M$ be any TM implementation. Let $e$ be an event applied by process $p$ to a base object $b$ as it performs a transaction $T$ during an execution $E$ of $M$. Let $E = \alpha \cdot e_1 \cdots e_m \cdot e \cdot \beta$ be an execution of $M$, where $\alpha$ and $\beta$ are execution fragments and $e_1 \cdots e_m$ is a maximal sequence of $m \geq 1$ consecutive nontrivial events by distinct distinct processes other than $p$ that access $b$. Then, we say that $T$ incurs *$m$ memory stalls in $E$ on account of $e$*. The *number of memory stalls incurred by $T$ in $E$* is the sum of memory stalls incurred by all events of $T$ in $E$ [7, 26].

**Theorem 4** ( [45])**.** *There exists a progressive, opaque and strict DAP TM implementation LP that provides wait-free TM-liveness, uses invisible reads, uses only read-write base objects, and for every execution $E$ and transaction $T_k \in txns(E)$:*

- *$T_k$ performs at most a single RAW, and*
- *every t-read operation invoked by $T_k$ incurs $O(1)$ memory stalls in $E$, and*
- *every complete t-read operation invoked by $T_k$ performs $O(|Rset(T_k)|)$ steps in $E$.*

**Proof sketch.** There exists a cheap progressive, opaque TM implementation *LP* in which every transaction performs at most a single RAW, every t-read operation incurs $O(1)$ memory stalls and maintains exactly one version of every t-object at any prefix of an execution. Moreover, the implementation is strict DAP and uses only read-write base objects.

For every t-object $X_j$, *LP* maintains a base object $v_j$ that stores the *value* of $X_j$. Additionally, for each $X_j$, we maintain a bit $L_j$, which if set, indicates the presence of an updating transaction writing to $X_j$. Also, for every process $p_i$ and t-object $X_j$, *LP* maintains a *single-writer bit* $r_{ij}$ to which only $p_i$ is allowed to write. Each of these base objects may be accessed only via read and write primitives.

The implementation first reads the value of t-object $X_j$ from base object $v_j$ and then reads the bit $L_j$ to detect contention with an updating transaction. If $L_j$ is set, the transaction is aborted; if not, read validation is performed on the entire read set. If the validation fails, the transaction is aborted. Otherwise, the implementation returns the value of $X_j$. For a read-only transaction $T_k$, $tryC_k$ simply returns the commit response.

The $write_k(X, v)$ implementation by process $p_i$ simply stores the value $v$ locally, deferring the actual updates to $tryC_k$. During $tryC_k$, process $p_i$ attempts to obtain exclusive write access to every $X_j \in Wset(T_k)$. This is realized through the single-writer bits, which ensure that no other transaction may write to base objects $v_j$ and $L_j$ until $T_k$ relinquishes its exclusive write access to $Wset(T_k)$. Specifically, process $p_i$ writes 1 to each $r_{ij}$, then checks that no other process $p_t$ has written 1 to any $r_{tj}$ by executing a series of reads (incurring a single RAW). If there exists such a process that concurrently contends on write set of $T_k$, for each $X_j \in Wset(T_k)$, $p_i$ writes 0 to $r_{ij}$ and aborts $T_k$. If successful in obtaining exclusive write access to

| TM-correctness | TM-liveness | Invisible reads | rmw primitives | Complexity |
|---|---|---|---|---|
| Strict serializability | WF | | read-write | Impossible |
| Strict serializability | | | read-write, conditional | $\Omega(n \log n)$ RMRs |
| Opacity | starvation-free | yes | read-write | $O(1)$ RAW/AWAR |

Table 2: Complexity bounds for strongly progressive TMs.

$Wset(T_k)$, $p_i$ sets the bit $L_j$ for each $X_j$ in its write set. Implementation of $tryC_k$ now checks if any t-object in its read set is concurrently contended by another transaction and then validates its read set. If there is contention on the read set or validation fails (indicating the presence of a conflicting transaction), the transaction is aborted. If not, $p_i$ writes the values of the t-objects to shared memory and relinquishes exclusive write access to each $X_j \in Wset(T_k)$ by writing 0 to each of the base objects $L_j$ and $r_{ij}$.

Read-only transactions do not apply any nontrivial primitives. Any updating transaction performs at most a single RAW in the course of acquiring exclusive write access to the transaction's write set. Thus, every transaction performs $O(1)$ non-overlapping RAWs in any execution.

Observe that a transaction may write to base objects $v_j$ and $L_j$ only after obtaining exclusive write access to t-object $X_j$, which in turn is realized via single-writer base objects. Thus, no transaction performs a write to any base object $b$ immediately after a write to $b$ by another transaction, *i.e.*, every transaction incurs only $O(1)$ memory stalls on account of any event it performs. The $read_k(X_j)$ implementation reads base objects $v_j$ and $L_j$, followed by the validation phase in which it reads $v_k$ for each $X_k$ in its current read set. Note that if the first read in the validation phase incurs a stall, then $read_k(X_j)$ aborts. It follows that each t-read incurs $O(1)$ stalls in every execution. □

## 3.3 Strongly progressive TMs

We then turn our focus to *strongly progressive* TMs [33] that, in addition to progressiveness, ensure that *not all* concurrent transactions conflicting over a single data item abort.

**A $\Omega(n \log n)$ lower bound on remote memory references.** [47] showed that in any *strongly progressive* strictly serializable TM implementation that accesses the shared memory with read, write and conditional primitives, such as compare-and-swap and load-linked/store-conditional, the total number of *remote memory references* (RMRs) that take place in an execution of a progressive TM in which

$n$ concurrent processes perform transactions on a single t-object might reach $\Omega(n \log n)$.

Modern shared memory CPU architectures employ a *memory hierarchy* [35]: a hierarchy of memory devices with different capacities and costs. Some of the memory is *local* to a given process while the rest of the memory is *remote*. Accesses to memory locations (*i.e.* base objects) that are *remote* to a given process are often orders of magnitude slower than a *local* access of the base object. Thus, the performance of concurrent implementations in the shared memory model may depend on the number of *remote memory references* made to base objects [6].

The RMR lower bound in [47] is obtained via a reduction to an analogous lower bound for mutual exclusion [10]. The reduction shows that any TM with the above properties can be used to implement a *deadlock-free* mutual exclusion, employing transactional operations on only one t-object and incurring a constant RMR overhead. The lower bound applies to RMRs in both the *cache-coherent (CC)* and *distributed shared memory (DSM)* models, and it appears to be the first RMR complexity lower bound for transactional memory.

**Theorem 5** ( [47])**.** *Any strictly serializable, strongly progressive TM implementation $M$ that accesses a single t-object implies a* deadlock-free*,* finite exit *mutual exclusion implementation $L(M)$ such that the RMR complexity of $M$ is within a constant factor of the RMR complexity of $L(M)$.*

**Strongly progressive TMs from read-write primitives.** Guerraoui *et al.* [33] proved the impossibility of implementing strongly progressive strictly serializable TMs providing *wait-free* TM-liveness from read-write base objects.

**Theorem 6** ( [33])**.** *It is impossible to implement strictly serializable strongly progressive TMs that provide* wait-free *TM-liveness (every t-operation returns a matching response within a finite number of steps) using only read and write primitives.*

[44] describes one means to circumvent this impossibility result: specifically, they prove the existence an opaque strongly progressive TM implementation from read-write base objects that provides starvation-free TM-liveness.

**Theorem 7.** *There exists a strongly progressive opaque TM implementation with starvation-free t-operations that uses invisible reads and employs at most four RAWs per transaction.*

## 3.4 On the cost of permissive opaque TMs

(Strongly) progressive TMs that allow a transaction to be aborted only on read-write conflicts have constant RAW/AWAR complexity. However, not aborting on

conflicts may not necessarily affect TM-correctness. Ideally, we would like to derive TM implementations that are *permissive* [30], in the sense that a transaction is aborted only if committing it would violate TM-correctness.

Kuznetsov *et al.* [44] establish a linear (in the transaction's data set size) separation between the worst-case transaction expensive synchronization complexity of strongly progressive TMs and *permissive* TMs that allow a transaction to abort only if committing it would violate opacity. Specifically, [44] show that an execution of a transaction in a *permissive opaque* TM implementation that provides starvation-free TM-liveness may require to perform at least one RAW/AWAR pattern *per* t-read.

**Definition 2** (Permissiveness)**.** *A TM implementation $M$ is* permissive with respect to TM-correctness $C$ *if for every history $H$ of $M$ such that $H$ ends with a response $r_k$ and replacing $r_k$ with some $r_k \neq A_k$ gives a history that satisfies $C$, we have $r_k \neq A_k$.*

Therefore, permissiveness does not allow a transaction to abort, unless committing it would violate the execution's correctness.

[44] show that an execution of a transaction in a *permissive opaque* TM implementation (providing starvation-free TM-liveness) may require to perform at least one RAW/AWAR pattern *per* t-read.

**Theorem 8** ( [44])**.** *Let $M$ be a permissive opaque TM implementation providing starvation-free TM-liveness. Then, for any $m \in \mathbb{N}$, $M$ has an execution in which some transaction performs $m$ t-reads such that the execution of each t-read contains at least one RAW or AWAR.*

*Proof.* Consider an execution $E$ of $M$ consisting of transactions $T_1$, $T_2$, $T_3$ as shown in Figure 3: $T_3$ performs a t-read of $X_1$, then $T_2$ performs a t-write on $X_1$ and commits, and finally $T_1$ performs a series of reads from objects $X_1, \ldots, X_m$. Since the implementation is permissive, no transaction can be forcefully aborted in $E$, and the only valid serialization of this execution is $T_3$, $T_2$, $T_1$. Note also that the execution generates a sequential history: each invocation of a t-operation is immediately followed by a matching response. Thus, since we assume starvation-freedom as a liveness property, such an execution exists.

We consider $read_1(X_k)$, $2 \leq k \leq m$ in execution $E$. Imagine that we modify the execution $E$ as follows. Immediately after $read_1(X_k)$ executed by $T_1$ we add $write_3(X, v)$, and $tryC_3$ executed by $T_3$ (let $TC_3(X_k)$ denote the complete execution of $W_3(X_k, v)$ followed by $tryC_3$). Obviously, $TC_3(X_k)$ must return abort: neither $T_3$ can be serialized before $T_1$ nor $T_1$ can be serialized before $T_3$. On the other hand if $TC_3(X_k)$ takes place just before $read_1(X_k)$, then $TC_3(X_k)$ must return commit but $read_1(X_k)$ must return the value written by $T_3$. In other words, $read_1(X_k)$ and

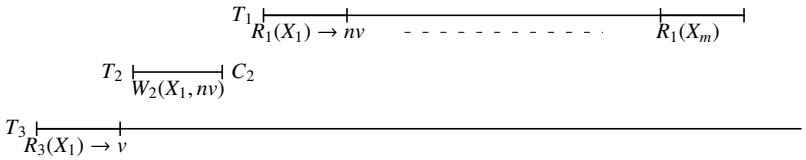



Figure 3: Execution $E$ of a permissive, opaque TM: $T_2$ and $T_3$ force $T_1$ to perform a RAW/AWAR in each $R_1(X_k)$, $2 \leq k \leq m$

$TC_3(X_k)$ are *strongly non-commutative* [8]: both of them see the difference when ordered differently. As a result, intuitively, $read_1(X_k)$ needs to perform a RAW or AWAR to make sure that the order of these two "conflicting" operations is properly maintained. We formalize this argument below.

Consider a modification $E'$ of $E$, in which $T_3$ performs $write_3(X_k)$ immediately after $read_1(X_k)$ and then tries to commit. In any serialization of $E'$, $T_3$ must precede $T_2$ ($read_3(X_1)$ returns the initial value of $X_1$) and $T_2$ must precede $T_1$ to respect the real-time order of transactions. The execution of $read_1(X_k)$ does not modify base objects, hence, $T_3$ does not observe $read_1(X_k)$ in $E'$. Since $M$ is permissive, $T_3$ must commit in $E'$. But since $T_1$ performs $read_1(X_k)$ before $T_3$ commits and $T_3$ updates $X_k$, we also have $T_1$ must precede $T_3$ in any serialization. Thus, $T_3$ cannot precede $T_1$ in any serialization—contradiction. Consequently, each $read_1(X_k)$ must perform a write to a base object.

Let $\pi$ be the execution fragment that represents the complete execution of $read_1(X_k)$ and $E^k$, the prefix of $E$ up to (but excluding) the invocation of $read_1(X_k)$.

Clearly, $\pi$ contains a write to a base object. Let $\pi_w$ be the first write to a base object in $\pi$. Thus, $\pi$ can be represented as $\pi_s \cdot \pi_w \cdot \pi_f$. Suppose that $\pi$ does not contain a RAW or AWAR. Consider the execution fragment $E^k \cdot \pi_s \cdot \rho$, where $\rho$ is the complete execution of $TC_3(X_k)$ by $T_3$. Such an execution of $M$ exists since $\pi_s$ does not perform any base object write, hence, $E^k \cdot \pi_s \cdot \rho$ is indistinguishable to $T_3$ from $E^k \cdot \rho$.

Since, by our assumption, $\pi_w \cdot \pi_f$ contains no RAW, any read performed in $\pi_w \cdot \pi_f$ can only be applied to base objects previously written in $\pi_w \cdot \pi_f$. Since $\pi_w$ is not an AWAR, $E^k \cdot \pi_s \cdot \rho \cdot \pi_w \cdot \pi_f$ is an execution of $M$ since it is indistinguishable to $T_1$ from $E^k \cdot \pi$. In $E^k \cdot \pi_s \cdot \rho \cdot \pi_w \cdot \pi_f$, $T_3$ commits (as in $\rho$) but $T_1$ ignores the value written by $T_3$ to $X_k$. But there exists no serialization that justifies this execution—contradiction to the assumption that $M$ is opaque. Thus, each $read_1(X_k)$, $2 \leq k \leq m$ must contain a RAW/AWAR.

Note that since all t-reads of $T_1$ are executed sequentially, all these RAW/AWAR patterns are pairwise non-overlapping, which completes the proof. □

The following result is a simple corollary to Theorem 8.

**Corollary 9** ( [16]). *There does not exist any permissive opaque TM implementation with invisible reads and starvation-free TM-liveness.*

# 4 Complexity of non-blocking TMs

We focus on TMs that avoid using locks and rely on non-blocking synchronization: a prematurely halted transaction cannot prevent other transactions from committing. Possibly the weakest non-blocking progress condition is obstruction-freedom [37, 40] stipulating that every transaction running in the absence of *step contention*, *i.e.*, not encountering steps of concurrent transactions, must commit. In fact, several early TM implementations [29, 38, 50, 61, 63] satisfied obstruction-freedom.

Let $\mathcal{OF}$ denote the class of non-blocking TMs that provide obstruction-free TM-progress (a transaction is allowed to abort only in executions that are not step contention-free) and *obstruction-free* (every t-operation must return a matching response within a finite number of steps in step contention-free executions) TM-liveness. Observe that there exists an execution exported by an obstruction-free TM, but not by any progressive TM and vice-versa. Consider a t-read $X$ by a transaction $T$ that runs step contention-free from a configuration that contains an incomplete write to $X$. Weak progressiveness does not preclude $T$ from being aborted in such an execution. Obstruction-free TMs however, must ensure that $T$ must complete its read of $X$ without blocking or aborting in such executions. On the other hand, weak progressiveness requires two non-conflicting transactions to not be aborted even in executions that are not step contention-free; but this is not guaranteed by obstruction-freedom.

## 4.1 Lower bounds for obstruction-free TMs

**On the cost of disjoint-access parallelism.** Complexity of obstruction-free TMs was first studied by Guerraoui and Kapalka [31, 33] who proved that they cannot provide strict DAP. However, it is possible to realize weaker than strict DAP variants of obstruction-free opaque TMs. For example, DSTM [38] satisfies RW DAP (and hence weak DAP), but not strict DAP.

**Theorem 10** ( [31]). *There does not exist any strict DAP strictly serializable TM implementation in $\mathcal{OF}$.*

The next result we survey focuses on strictly serializable TM implementations that satisfy two important properties: weak DAP and read invisibility. There exist weak DAP lock-based TM implementations that use invisible reads [21, 27]. In contrast, [45] establish that it is impossible to implement an obstruction-free

**Algorithm 1** Strict DAP progressive opaque TM implementation *LP*; code for $T_k$ executed by process $p_i$

1: **Shared base objects:**
2: $v_j$, for each t-object $X_j$
3: $r_{ij}$, for each process $p_i$ and t-object $X_j$
4: single-writer bit
5: $L_j$, for each t-object $X_j$

6: ***read***$_k(X_j)$**:**
7: **if** $X_j \notin Rset(T_k)$ **then**
8: $[ov_j, k_j] := read(v_j)$
9: $Rset(T_k) := Rset(T_k) \cup \{X_j, [ov_j, k_j]\}$
10: **if** $read(L_j) \neq 0$ **then**
11: **Return** $A_k$
12: **if** validate() **then**
13: **Return** $A_k$
14: **Return** $ov_j$
15: **else**
16: $[ov_j, \perp] := Rset(T_k)$.locate$(X_j)$
17: **Return** $ov_j$

18: ***write***$_k(X_j, v)$**:**
19: $nv_j := v$
20: $Wset(T_k) := Wset(T_k) \cup \{X_j\}$
21: **Return** *ok*

22: ***tryC***$_k()$**:**
23: **if** $|Wset(T_k)| = \emptyset$ **then**
24: **Return** $C_k$
25: locked := acquire$(Wset(T_k))$
26: **if** $\neg$ locked **then**
27: **Return** $A_k$
28: **if** isAbortable() **then**
29: release$(Wset(T_k))$
30: **Return** $A_k$
31: **for all** $X_j \in Wset(T_k)$ **do**
32: $write(v_j, [nv_j, k])$
33: release$(Wset(T_k))$
34: **Return** $C_k$

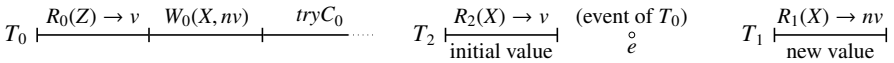
35: **Function:** release$(Q)$**:**
36: **for all** $X_j \in Q$ **do**
37: $write(L_j, 0)$
38: **for all** $X_j \in Q$ **do**
39: $write(r_{ij}, 0)$
40: **Return** *ok*

41: **Function:** acquire$(Q)$**:**
42: **for all** $X_j \in Q$ **do**
43: $write(r_{ij}, 1)$
44: **if** $\exists X_j \in Q; t \neq k : read(r_{tj}) = 1$ **then**
45: **for all** $X_j \in Q$ **do**
46: $write(r_{ij}, 0)$
47: **Return** *false*
48: **for all** $X_j \in Q$ **do**
49: $write(L_j, 1)$
50: **Return** *true*

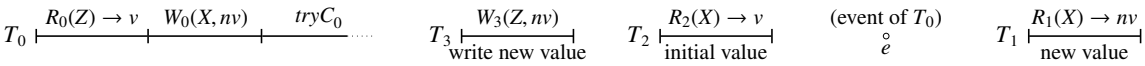
51: **Function:** isAbortable() **:**
52: **if** $\exists X_j \in Rset(T_k) : X_j \notin Wset(T_k) \wedge read(L_j) \neq 0$ **then**
53: **Return** *true*
54: **if** validate() **then**
55: **Return** *true*
56: **Return** *false*

57: **Function:** validate() **:**
58: **if** $\exists X_j \in Rset(T_k)$:$[ov_j, k_j] \neq read(v_j)$ **then**
59: **Return** *true*
60: **Return** *false*


(a) $T_1$ must read the base object updated in $e$ and return the new value $nv$ of $X$




(b) $T_1$ returns new value of $X$ since $T_2$ is invisible




(c) By weak DAP and invisible reads, $T_1$ and $T_2$ do not observe the presence of $T_3$

Figure 4: Executions describing the proof sketch of Theorem 11; execution in 4c is not strictly serializable

TM that provides both weak DAP and read invisibility. Indeed, DSTM [38] and FSTM [29] are weak DAP, but use *visible* reads for aborting pending writing transactions.

**Theorem 11** ( [45]). *There does not exist a weak DAP strictly serializable TM implementation in $\mathcal{OF}$ that uses invisible reads.*

**Proof sketch.** Suppose, by contradiction, that such a TM implementation $M$ exists. Consider an execution $E$ of $M$ in which a transaction $T_0$ performs a t-read of t-object $Z$ (returning the initial value $v$), writes $nv$ (new value) to t-object $X$, and commits. Let $E'$ denote the longest prefix of $E$ that cannot be extended with the t-complete step contention-free execution of any transaction that reads $nv$ in $X$ and commits.

Thus if $T_0$ takes one more step, then the resulting execution $E' \cdot e$ can be extended with the t-complete step contention-free execution of a transaction $T_1$ that reads $nv$ in $X$ and commits.

Since $M$ uses invisible reads, the following execution exists: $E'$ can be extended with the t-complete step contention-free execution of a transaction $T_2$ that reads the initial value $v$ in $X$ and commits, followed by the step $e$ of $T_0$ after which transaction $T_1$ running step contention-free reads $nv$ in $X$ and commits. Moreover, this execution is indistinguishable to $T_1$ and $T_2$ from an execution in which the read set of $T_0$ is empty. Thus, we can modify this execution by inserting the step contention-free execution of a committed transaction $T_3$ that writes a new value to $Z$ after $E'$, but preceding $T_2$ in real-time order. Intuitively, by weak DAP,

transactions $T_1$ and $T_2$ cannot distinguish this execution from the original one in which $T_3$ does not participate.

Thus, we can show that the following execution exists: $E'$ is extended with the t-complete step contention-free execution of $T_3$ that writes $nv$ to $Z$ and commits, followed by the t-complete step contention-free execution of $T_2$ that reads the initial value $v$ in $X$ and commits, followed by the step $e$ of $T_0$, after which $T_1$ reads $nv$ in $X$ and commits.

This execution is, however, not strictly serializable: $T_0$ must appear in any serialization ($T_1$ reads a value written by $T_0$). Transaction $T_2$ must precede $T_0$, since the t-read of $X$ by $T_2$ returns the initial value of $X$. To respect real-time order, $T_3$ must precede $T_2$. Finally, $T_0$ must precede $T_3$ since the t-read of $Z$ returns the initial value of $Z$. The cycle $T_0 \to T_3 \to T_2 \to T_0$ implies a contradiction. □

**A linear lower bound on memory stall complexity.** [45] prove a linear (in $n$) lower bound for strictly serializable TM implementations in $\mathcal{OF}$ on the total number of *memory stalls* incurred by a single t-read operation.

**Theorem 12** ( [45])**.** *Every strictly serializable TM implementation $M \in \mathcal{OF}$ has a $(n-1)$-stall execution $E$ for a t-read operation performed in $E$.*

**Proof sketch.** Inductively, for each $k \leq n-1$, construct a specific *$k$-stall execution* [26] in which some t-read operation by a process $p$ incurs $k$ stalls. In the $k$-stall execution, $k$ processes are partitioned into disjoint subsets $S_1, \ldots, S_i$. The execution can be represented as $\alpha \cdot \sigma_1 \cdots \sigma_i$; $\alpha$ is $p$-free, where in each $\sigma_j$, $j = 1, \ldots, i$, $p$ first runs by itself, then each process in $S_j$ applies a *nontrivial* event on a base object $b_j$, and then $p$ applies an event on $b_j$. Moreover, $p$ does not detect step contention in this execution and, thus, must return a non-abort value in its t-read and commit in the solo extension of it. Additionally, it is guaranteed that in any extension of $\alpha$ by the processes other than $\{p\} \cup S_1 \cup S_2 \cup \ldots \cup S_i$, no nontrivial primitive is applied on a base object accessed in $\sigma_1 \cdots \sigma_i$.

Assuming that $k \leq n-2$, we introduce a not previously used process executing an updating transaction immediately after $\alpha$, so that the subsequent t-read operation executed by $p$ is "perturbed" (must return another value). This will help us to construct a $(k+k')$-stall execution $\alpha \cdot \alpha' \cdot \sigma_1 \cdots \sigma_i \cdot \sigma_{i+1}$, where $k' > 0$. □

Observe that, since there are at most $n$ concurrent transactions, we cannot do better than $(n-1)$ stalls. Thus, the lower bound of Theorem 12 is tight.

**RAW/AWAR complexity.** [45] prove that opaque, RW DAP TM implementations in $\mathcal{OF}$ have executions in which some read-only transaction performs a linear (in $n$) number of non-overlapping RAWs or AWARs.

| | Obstruction-free | Progressive *LP* |
|---|---|---|
| strict DAP | No | Yes |
| invisible reads+weak DAP | No | Yes |
| stall complexity of reads | $\Omega(n)$ | $O(1)$ |
| RAW/AWAR complexity | $\Omega(n)$ | $O(1)$ |
| read-write base objects, wait-free termination | No | Yes |

Figure 5: Complexity gap between blocking and non-blocking TMs; $n$ is the number of processes

**Theorem 13.** *Every RW DAP opaque TM implementation $M \in \mathcal{OF}$ has an execution $E$ in which some read-only transaction $T \in txns(E)$ performs $\Omega(n)$ non-overlapping RAW/AWARs.*

**Impossibility of obstruction-free TMs from read-write primitives.** Guerraoui and Kapalka [31, 33] also proved that a strict serializable TM that provides OF TM-progress and wait-free TM-liveness cannot be implemented using only read and write primitives. An interesting open question is whether we can implement strict serializable TMs in $\mathcal{OF}$ using only read and write primitives.

## 4.2 Blocking versus non-blocking TMs

Some benefits of obstruction-free TMs, namely their ability to make progress even if some transactions prematurely fail, are not provided by progressive TMs. However, several papers [20,21,28] argued that lock-based TMs tend to outperform obstruction-free ones by allowing for simpler algorithms with lower overhead, and their inherent progress issues may be resolved using timeouts and *contention-managers* [60].

As highlighted in [21,28], obstruction-free TMs typically must forcefully abort pending conflicting transactions. This observation inspires the impossibility of invisible reads (Theorem 11). Typically, to detect the presence of a conflicting transaction and abort it, the reading transaction must employ a RAW or a read-modify-write primitive like *compare-and-swap*, motivating the linear lower bound on expensive synchronization (Theorem 13). Also, in obstruction-free TMs, a transaction may not wait for a concurrent inactive transaction to complete and, as a result, we may have an execution in which a transaction incurs a distinct stall due to a transaction run by each other process, hence the linear stall complexity (Theorem 12). Intuitively, since transactions in progressive TMs may abort themselves in case of conflicts, they can employ invisible reads and maintain constant stall and RAW/AWAR complexities.

Overcoming the lower bounds for obstruction-free TMs individually is comparatively easy. Say, TL [21] combines strict DAP with invisible reads, but it

is not read-write (for base object primitives), and it does not provide constant RAW/AWAR and stall complexities. However, the progressive TM *LP* overcomes most of the lower bounds known for obstruction-free TMs. Observe that the opaque implementation *LP*, (1) uses only read-write base objects and ensures that every transactional operation terminates in a wait-free manner, (2) ensures strict DAP, (3) has invisible reads, (4) performs $O(1)$ non-overlapping RAWs/AWARs per transaction, and (5) incurs $O(1)$ memory stalls per read operation. In contrast, from the lower bounds summarized in this survey we know that (i) no OF TM that provides wait-free transactional operations can be implemented using only read-write base objects; (ii) no OF TM can provide strict DAP; (iii) no weak DAP OF TM has invisible reads and (iv) no OF TM ensures a constant number of stalls incurred by a read operation. Finally, (v) no RW DAP *opaque* OF TM has constant RAW/AWAR complexity. Thus, (iv) and (v) exhibit a linear separation between blocking and non-blocking TMs w.r.t expensive synchronization and memory stall complexity, respectively.

The results are summarized and put in perspective in Figure 5 [45]. Altogether, we grasp a considerable complexity gap between blocking and non-blocking TM implementations, justifying theoretically the shift in TM practice we observed during the past decade.

# 5 Lower bounds for partially non-blocking TMs

It is easy to see that *dynamic* TMs where the patterns in which transactions access t-objects are not known in advance do not allow for *wait-free* TMs [33], *i.e.*, every transaction must commit in a finite number of steps of the process executing it, regardless of the behavior of concurrent processes. Suppose that a transaction $T_1$ reads t-object $X$, then a concurrent transaction $T_2$ reads t-object $Y$, writes to $X$ and commits, and finally $T_2$ writes to $Y$. Since $T_1$ has read the "old" value in $X$ and $T_2$ has read the "old" value in $Y$, there is no way to commit $T_1$ and order the two transactions in a sequential execution. As this scenario can be repeated arbitrarily often, even the weaker guarantee of *local progress* that only requires that each transaction *eventually* commits if repeated sufficiently often, cannot be ensured by *any* strictly serializable TM implementation, regardless of the base objects it uses [14].[1]

**Theorem 14** ( [14])**.** *There does not exist any strictly serializable TM implementation that provides local progress.*

[1]Note that the counter-example would not work if we imagine that the data sets accessed by a transaction can be known in advance. However, we consider the conventional dynamic TM programming model.

But can we ensure that at least *some* transactions commit wait-free and what are the inherent costs? It is often argued that many realistic workloads are *read-dominated*: the proportion of read-only transactions is higher than that of updating ones, or read-only transactions have much larger data sets than updating ones [12, 34]. Therefore, it seems natural to require that read-only transactions commit wait-free. Moreover, we require that updating transaction provide only an extremely weak sequential TM-progress.

We denote by $\mathcal{RWF}$ the class of partially non-blocking TMs originally studied and motivated by Attiya *et al.* [11].

**Definition 3** ( [46]). *(The class $\mathcal{RWF}$) A TM implementation $M \in \mathcal{RWF}$* iff *in its every execution:*

- *(*wait-free TM-progress for read-only transactions*) every read-only transaction commits in a finite number of its steps, and*
- *(*sequential TM-progress and sequential TM-liveness for updating transactions*)* i.e.*, every transaction running step contention-free from a t-quiescent configuration, commits in a finite number of its steps.*

## 5.1 The space complexity of invisible reads

[46] prove that every strictly serializable TM implementation $M \in \mathcal{RWF}$ that uses invisible reads must keep unbounded sets of values for every t-object. To do so, for every $c \in \mathbb{N}$, construct an execution of $M$ that *maintains at least c distinct values for every t-object*.

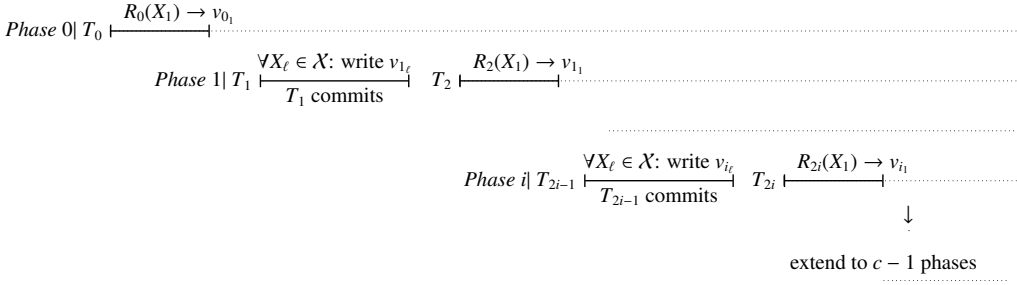
**Definition 4** ( [46]). *Let $E$ be any execution of a TM implementation $M$. We say that $E$* maintains $c$ distinct values $\{v_1, \ldots, v_c\}$ of t-object $X$, *if there exists an execution $E \cdot E'$ of $M$ such that*

- *$E'$ contains the complete executions of $c$ t-reads of $X$ and,*
- *for all $i \in \{1, \ldots, c\}$, the response of the $i^{th}$ t-read of $X$ in $E'$ is $v_i$.*

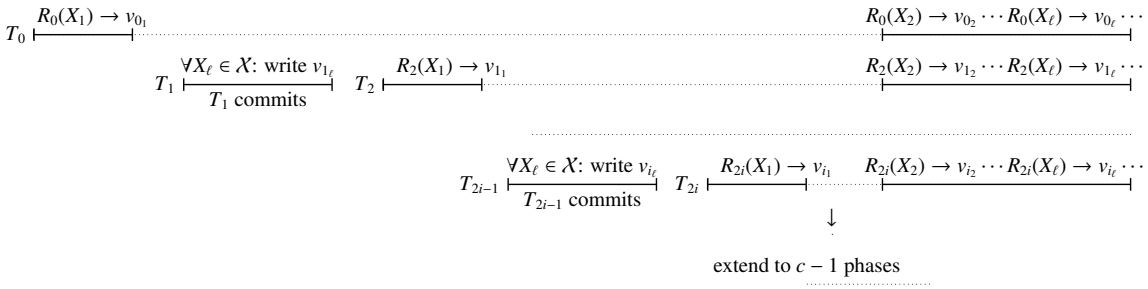
**Theorem 15** ( [46]). *Let $M$ be any strictly serializable TM implementation in $\mathcal{RWF}$ that uses invisible reads, and $\mathcal{X}$, any set of t-objects. Then, for every $c \in \mathbb{N}$, there exists an execution $E$ of $M$ such that $E$ maintains at least $c$ distinct values of each t-object $X \in \mathcal{X}$.*

*Proof.* Let $v_{0_\ell}$ be the initial value of t-object $X_\ell \in \mathcal{X}$. For every $c \in \mathbb{N}$, we iteratively construct an execution $E$ of $M$ of the form depicted in Figure 6a. The construction of $E$ proceeds in phases: there are at most $c - 1$ phases. For all $i \in \{0, \ldots c - 1\}$, we denote the execution after phase $i$ as $E_i$ which is defined as follows:

- $E_0$ is the complete step contention-free execution fragment $\alpha_0$ of read-only transaction $T_0$ that performs $read_0(X_1) \to v_{0_1}$




(a) for all $i \in \{1, \ldots, c-1\}$, $T_{2i-1}$ writes $v_{i_\ell}$ to each $X_\ell$; $read_{2i}(X_1)$ must return $v_{i_1}$




(b) extend every read-only transaction $T_{2i}$ in phase $i$ with t-reads of $X_2, \ldots X_\ell, \ldots$; note that each $read_{2i}(X_\ell)$ must return $v_{i_\ell}$

Figure 6: Executions in the proof of Theorem 15; execution in 6a must maintain $c$ distinct values of every t-object

- for all $i \in \{1, \ldots, c-1\}$, $E_i$ is defined to be an execution of the form $\alpha_0 \cdot \rho_1 \cdot \alpha_1 \cdots \rho_i \cdot \alpha_i$ such that for all $j \in \{1, \ldots, i\}$,
  - $\rho_j$ is the t-complete step contention-free execution fragment of an updating transaction $T_{2j-1}$ that, for all $X_\ell \in \mathcal{X}$ writes the value $v_{j_\ell}$ and commits
  - $\alpha_j$ is the complete step contention-free execution fragment of a read-only transaction $T_{2j}$ that performs $read_{2j}(X_1) \to v_{j_1}$

Since read-only transactions are invisible, for all $i \in \{0, \ldots, c-1\}$, the execution fragment $\alpha_i$ does not contain any nontrivial events. Consequently, for all $i < j \leq c-1$, the configuration after $E_i$ is indistinguishable to transaction $T_{2j-1}$ from a t-quiescent configuration and it must be committed in $\rho_j$ (by sequential progress for updating transactions). Observe that, for all $1 \leq j < i$, $T_{2j-1} \prec_E^{RT} T_{2i-1}$. Strict serializability of $M$ now stipulates that, for all $i \in \{1, \ldots, c-1\}$, the t-read of $X_1$ performed by transaction $T_{2i}$ in the execution fragment $\alpha_i$ must return the value $v_{i_1}$ of $X_1$ as written by transaction $T_{2i-1}$ in the execution fragment $\rho_i$ (in any serialization, $T_{2i-1}$ is the latest committed transaction writing to $X_1$ that precedes $T_{2i}$). Thus, $M$ indeed has an execution $E$ of the form depicted in Figure 6a.

Consider the execution fragment $E'$ that extends $E$ in which, for all $i \in \{0, \ldots, c-1\}$, read-only transaction $T_{2i}$ is extended with the complete execution of the t-reads of every t-object $X_\ell \in \mathcal{X} \setminus \{X_1\}$ (depicted in Figure 6b).

We claim that, for all $i \in \{0, \ldots, c-1\}$, and for all $X_\ell \in \mathcal{X} \setminus \{X_1\}$, $read_{2i}(X_\ell)$ performed by transaction $T_{2i}$ must return the value $v_{i_\ell}$ of $X_\ell$ written by transaction $T_{2i-1}$ in the execution fragment $\rho_i$. Indeed, by wait-free progress, $read_i(X_\ell)$ must return a non-abort response in such an extension of $E$. Suppose by contradiction that $read_i(X_\ell)$ returns a response that is not $v_{i_\ell}$. There are two cases:

- $read_{2i}(X_\ell)$ returns the value $v_{j_\ell}$ written by transaction $T_{2j-1}$; $j < i$. However, since for all $j < i$, $T_{2j} \prec_E^{RT} T_{2i}$, the execution is not strictly serializable—contradiction.
- $read_{2i}(X_\ell)$ returns the value $v_{j_\ell}$ written by transaction $T_{2j}$; $j > i$. Since $read_i(X_1)$ returns the value $v_{i_1}$ and $T_{2i} \prec_E^{RT} T_{2j}$, there exists no such serialization—contradiction.

Thus, $E$ maintains at least $c$ distinct values of every t-object $X \in \mathcal{X}$. □

Perelman *et al.* [55] considered the closely related (to $\mathcal{RWF}$) class of *mv-permissive* TMs: a transaction can only be aborted if it is an updating transaction that conflicts with another updating transaction. $\mathcal{RWF}$ is incomparable with the class of mv-permissive TMs. On the one hand, mv-permissiveness guarantees that read-only transactions never abort, but does not imply that they commit in a wait-free manner. On the other hand, $\mathcal{RWF}$ allows an updating transaction to abort in the presence of a concurrent read-only transaction, which is disallowed by mv-permissive TMs. Observe that, technically, mv-permissiveness is a blocking TM-progress condition, although when used in conjunction with wait-free TM-liveness, it is a partially non-blocking TM-progress condition that is strictly stronger than $\mathcal{RWF}$.

[55] proved that mv-permissive TMs cannot be *online space optimal*, *i.e.*, no mv-permissive TM can keep the minimum number of old object versions for any TM history. The result on the space complexity of implementations in $\mathcal{RWF}$ that use invisible reads (Theorem 15) is different since it proves that the implementation must maintain an unbounded number of versions of every t-object. The above proof technique can however be used to show that mv-permissive TMs considered in [55] should also maintain unbounded number of versions.

## 5.2 On the cost of disjoint-access parallelism

Kuznetsov *et al.* [46] prove that it is impossible to derive strictly serializable TM implementations in $\mathcal{RWF}$ which ensure that any two transactions accessing pairwise disjoint data sets can execute without contending on the same base object.

**Theorem 16** ( [46]). *There exists no strictly serializable strict DAP TM implementation in $\mathcal{RWF}$.*

Kuznetsov *et al.* [46] also prove a linear lower bound (in the size of the transaction's read set) on the number of RAWs or AWARs for weak DAP TM implementations in $\mathcal{RWF}$. Specifically, there exist executions in which each t-read operation of an arbitrarily long read-only transaction contains a RAW or an AWAR.

**Theorem 17** ( [46]). *Every strictly serializable weakly DAP TM implementation $M \in \mathcal{RWF}$ has, for all $m \in \mathbb{N}$, an execution in which some read-only transaction $T_0$ with $m = |Rset(T_0)|$ performs $\Omega(m)$ RAWs/AWARs.*

Since Theorem 17 implies that read-only transactions must perform nontrivial events, we have the following corollary that was proved directly in [11].

**Corollary 18** ( [11]). *There does not exist any strictly serializable weak DAP TM implementation $M \in \mathcal{RWF}$ that uses invisible reads.*

Attiya et al. [11] also considered a stronger "disjoint-access" property, called simply DAP, referring to the original definition proposed Israeli and Rappoport [42]. In DAP, two transactions are allowed to *concurrently access* (even for reading) the same base object only if they are disjoint-access. For an $n$-process DAP TM implementation, it is shown in [11] that a read-only transaction must perform at least $n - 3$ writes. The lower bound in Theorem 17 is strictly stronger than the one in [11], as it assumes only weak DAP, considers a more precise RAW/AWAR metric, and does not depend on the number of processes in the system. (Technically, the last point follows from the fact that the execution constructed in the proof of Theorem 17 uses only 3 concurrent processes.) Thus, the theorem subsumes the two lower bounds of [11] within a single proof.

Assuming starvation-free TM-liveness, [55] showed that implementing a weak DAP strictly serializable mv-permissive TM is impossible. The proof of this result is immediate from the analogous results for $\mathcal{RWF}$ in [11] and [46].

# 6 Hybrid Transactional Memory

If used carefully, HTM can be an extremely useful construct, and can significantly speed up and simplify concurrent implementations. At the same time, this powerful tool is not without its limitations: since HTMs are usually implemented on top of the cache coherence mechanism, hardware transactions have inherent *capacity constraints* on the number of distinct memory locations that can be accessed inside a single transaction. Moreover, all current proposals are *best-effort*, as they may

abort under imprecisely specified conditions (cache capacity overflow, interrupts *etc*). In brief, the programmer should not solely rely on HTMs.

Several HyTM schemes [17, 19, 43, 48] have been proposed to complement the fast, but best-effort nature of HTM with a slow, reliable software transactional memory (STM) backup. These proposals have explored a wide range of trade-offs between the overhead on hardware transactions, concurrent execution of hardware and software, and the provided progress guarantees. Early proposals for HyTM implementations [19, 43] shared some interesting features. First, transactions that do not conflict are expected to run concurrently, regardless of their types (software or hardware), à la progressiveness. Second, in addition to changing the values of transactional objects, hardware transactions usually employ *code instrumentation* techniques. Intuitively, instrumentation is used by hardware transactions to detect concurrency scenarios and abort in the case of contention. The number of instrumentation steps performed by these implementations within a hardware transaction is usually proportional to the size of the transaction's data set.

Recent work by Riegel *et al.* [58] surveyed the various HyTM algorithms to date, focusing on techniques to reduce instrumentation overheads in the frequently executed hardware fast-path. However, it is not clear whether there are fundamental limitations when building a HyTM with non-trivial concurrency between hardware and software transactions. In particular, what are the inherent instrumentation costs of building a HyTM, and what are the trade-offs between these costs and the provided *concurrency*, *i.e.*, the ability of the HyTM system to run software and hardware transactions in parallel?

**Modelling HyTM.** To address these questions, [4] proposes a model for hybrid TM systems which formally captures the notion of *cached* accesses provided by hardware transactions, and precisely defines instrumentation costs in a quantifiable way. [4] models a hardware transaction as a series of memory accesses that operate on locally cached copies of the variables, followed by a *cache-commit* operation. In case a concurrent transaction performs a (read-write or write-write) conflicting access to a cached object, the cached copy is invalidated and the hardware transaction aborts. The model for instrumentation is motivated by recent experimental evidence which suggests that the overhead on hardware transactions imposed by code which detects concurrent software transactions is a significant performance bottleneck [51]. In particular, a HyTM implementation imposes a logical partitioning of shared memory into *data* and *metadata* locations. Intuitively, metadata is used by transactions to exchange information about contention and conflicts while data locations only store the *values* of data items read and updated within transactions. [4] quantifies instrumentation cost by measuring the number of accesses to *metadata objects* which transactions perform. All known HyTM

proposals, such as *HybridNOrec* [17, 57], *PhTM* [48] and others [19, 43] avoid co-locating the data and metadata within a single base object.

**Complexity.** Once this general model is in place, Alistarh *et al.* [4] derive two lower bounds on the cost of implementing a HyTM. First, they show that some instrumentation is necessary in a HyTM implementation even if we only intend to provide *sequential* progress, where any transaction is only guaranteed to commit if it runs in the absence of concurrency.

**Theorem 19** ( [4])**.** *There does not exist a strictly serializable uninstrumented HyTM implementation that ensures sequential TM-progress and TM-liveness.*

Second, [4] prove that any progressive HyTM implementation providing *obstruction-free liveness* (every operation running *solo* returns some response) and has executions in which an arbitrarily long read-only hardware transaction running in the absence of concurrency *must* access a number of distinct metadata objects proportional to the size of its data set.

**Theorem 20** ( [4])**.** *Let $\mathcal{M}$ be any progressive, opaque HyTM implementation that provides OF TM-liveness. For every $m \in \mathbb{N}$, there exists an execution $E$ in which some fast-path read-only transaction $T_k \in txns(E)$ satisfies either (1) $Dset(T_k) \leq m$ and $T_k$ incurs a capacity abort in $E$ or (2) $Dset(T_k) = m$ and $T_k$ accesses $\Omega(m)$ distinct metadata base objects in $E$.*

The proof of the above theorem proceeds inductively. Start with a sequential execution in which a "large" set $S_m$ of read-only hardware transactions, each accessing $m$ distinct data items and $m$ distinct metadata memory locations, run after an execution $E_m$. We then construct execution $E_{m+1}$, an extension of $E_m$, which forces at least half of the transactions in $S_m$ to access a *new* metadata base object when reading a new $(m+1)^{th}$ data item, running after $E_{m+1}$. The technical challenge, and the key departure from prior work on STM lower bounds, *e.g.* [11, 31, 33], is that hardware transactions practically possess "automatic" conflict detection, aborting on contention. This is in contrast to STMs, which must take steps to detect contention on memory locations.

**Algorithms.** The inherent high instrumentation costs of early HyTM designs, stimulated more recent HyTM schemes [17, 48, 51, 58] to sacrifice progressiveness for *constant* instrumentation cost (*i.e.*, not depending on the size of the transaction). In the past few years, Dalessandro *et al.* [17] and Riegel *et al.* [58] have proposed HyTMs based on the efficient NOrec STM [18]. These HyTMs schemes do not guarantee any parallelism among transactions; only sequential progress is ensured. Despite this, they are among the best-performing HyTMs to date due to the limited

instrumentation in hardware transactions. Therefore, the cost of avoiding the linear lower bound for progressive implementations is that hardware transactions may be aborted by non-conflicting software ones.

# 7 Research directions and open questions

**Weak TM-correctness.** In this survey, we focussed on TM implementations providing the TM-correctness properties of opacity or the weaker strict serializability. However, one may observe that as long as committed transactions constitute a serial execution and every transaction witnesses a consistent state, the execution can be considered "safe": no run-time error that cannot occur in a serial execution can happen. TM-correctness properties like virtual-world consistency (VWC) [41] and transactional memory specification (TMS1) [25] ensure strict serializability, but are strictly weaker than opacity. Are TM implementations that satisfy VWC or TMS1, but not opacity subject to the lower bounds surveyed in this paper? For instance, it is easy to see that the lower bound of Theorem 8 on the complexity of permissive opaque TMs is not subject to permissive VWC TMs [16]. Furthermore, [16] described a permissive VWC TM implementation that ensures that t-read operations do not perform nontrivial primitives, but the tryCommit invoked by a read-only transaction perform a linear (in the size of the transaction's data set) number of RAW/AWARs.

Bushkov et al. [13] improved on the impossibility result in [31] and showed that a variant of strict DAP cannot be combined with obstruction-free TM-progress, even if a weaker (than strictly serializability) TM-correctness property is assumed.

Peluso et al. [54] study the complexity of TM implementations in the class $\mathcal{RWF}$ and show that deriving DAP implementations is impossible even if the TM-correctness assumed is weaker than strict serializability.

Exploring the complexity of STM and HyTM implementations satisfying the TM-correctness properties of VWC and TMS1 as well as properties weaker than strict serializability opens up several open questions and research directions.

**HyTM models and complexity.** Recent work has investigated alternatives to HyTMs that rely on STM fallback, such as *sandboxing* [3, 15] or *hardware-accelerated* STM [59,62], and the use of both direct *and* cached accesses within the same hardware transaction to reduce instrumentation overhead [57, 58]. Another recent approach proposed *reduced hardware transactions* [51], where a part of the slow-path is executed using a short fast-path transaction, which allows to partially eliminate instrumentation from the hardware fast-path.

Verifying the correctness and understanding the complexity of these protocols is an important research direction as is identifying techniques for automatically

deploying the best TM implementation for a given workload [22] and scheduling techniques for HyTMs [23].

# References


[1] Advanced Synchronization Facility Proposed Architectural Specification, March 2009. `http://developer.amd.com/wordpress/media/2013/09/45432-ASF_Spec_2.1.pdf`.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] Y. Afek, A. Levy, and A. Morrison. Software-improved hardware lock elision. In *PODC*. ACM, 2014.

[4] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 185–199, 2015.

[5] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, Oct. 1985.

[6] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[7] H. Attiya, R. Guerraoui, D. Hendler, and P. Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4), 2009.

[8] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[9] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety and deferred update in transactional memory. In R. Guerraoui and P. Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *Lecture Notes in Computer Science*, pages 50–71. Springer International Publishing, 2015.

[10] H. Attiya, D. Hendler, and P. Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 447–447, New York, NY, USA, 2008. ACM.

[11] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4):698–719, 2011.

[12] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. The pcl theorem: Transactions cannot be parallel, consistent and live. In *SPAA*, pages 178–187, 2014.

[14] V. Bushkov, R. Guerraoui, and M. Kapalka. On the liveness of transactional memory. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 9–18, New York, NY, USA, 2012. ACM.

[15] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop*. ACM, 2014.

[16] T. Crain, D. Imbs, and M. Raynal. Read invisibility, virtual world consistency and permissiveness are compatible. Research Report, ASAP - INRIA - IRISA - CNRS : UMR6074 - INRIA - Institut National des Sciences Appliquées de Rennes - Université de Rennes I, 11 2010.

[17] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 39–52. ACM, 2011.

[18] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. *SIGPLAN Not.*, 45(5):67–78, Jan. 2010.

[19] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006.

[20] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

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

[22] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano. Proteustm: Abstraction meets performance in transactional memory. *SIGOPS Oper. Syst. Rev.*, 50(2):757–771, Mar. 2016.

[23] N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 224–233, New York, NY, USA, 2015. ACM.

[24] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 216–224, New York, NY, USA, 2004. ACM.

[25] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.

[26] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[27] R. Ennals. The lightweight transaction library. http://sourceforge.net/projects/libltx/files/.

[28] R. Ennals. Software transactional memory should not be obstruction-free. 2005.

[29] K. Fraser. Practical lock-freedom. Technical report, Cambridge University Computer Laborotory, 2003.

[30] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, 2008.

[31] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.

[32] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44(1):404–415, Jan. 2009.

[33] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[34] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, Mar. 2007.

[35] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[36] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):123–149, 1991.

[37] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, pages 522–529, 2003.

[38] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[39] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[40] M. Herlihy and N. Shavit. On the nature of progress. In *OPODIS*, pages 313–328, 2011.

[41] D. Imbs and M. Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444, July 2012.

[42] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, pages 151–160, 1994.

[43] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[44] P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011.

[45] P. Kuznetsov and S. Ravi. Grasping the gap between blocking and non-blocking transactional memories. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 232–247, 2015.

[46] P. Kuznetsov and S. Ravi. On partial wait-freedom in transactional memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, page 10, 2015.

[47] P. Kuznetsov and S. Ravi. Progressive transactional memory in time and space. In *Parallel Computing Technologies - 13th International Conference, PaCT 2015, Petrozavodsk, Russia, August 31 - September 4, 2015, Proceedings*, pages 410–425, 2015.

[48] Y. Lev, M. Moir, and D. Nussbaum. Phtm: Phased transactional memory. In *In Workshop on Transactional Computing (Transact), 2007. research.sun.com/scalable/pubs/TRANSACT2007PhTM.pdf*.

[49] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[50] V. J. Marathe, W. N. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[51] A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

[52] P. E. McKenney. Memory barriers: a hardware view for software hackers. Linux Technology Center, IBM Beaverton, June 2010.

[53] M. Ohmacht. Memory Speculation of the Blue Gene/Q Compute Chip, 2011. `http://wands.cse.lehigh.edu/IBM_BQC_PACT2011.ppt`.

[54] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 217–226, New York, NY, USA, 2015. ACM.

[55] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, 2010.

[56] J. Reinders. Transactional Synchronization in Haswell, 2012. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`.

[57] T. Riegel. Software Transactional Memory Building Blocks. 2013.

[58] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 53–64. ACM, 2011.

[59] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[60] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[61] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[62] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 210–220, New York, NY, USA, 2007. ACM.

[63] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. Nztm: Nonblocking zero-indirection transactional memory. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 204–213, New York, NY, USA, 2009. ACM.