Theory Comput Syst (2011) 49:698–719
DOI 10.1007/s00224-010-9304-5

Inherent Limitations on Disjoint-Access Parallel Implementations of Transactional Memory

Hagit Attiya · Eshcar Hillel · Alessia Milani

Published online: 2 December 2010

Abstract Transactional memory (TM) is a popular approach for alleviating the difficulty of programming concurrent applications; TM guarantees that a *transaction*, consisting of a sequence of operations, appear to be executed atomically. Two fundamental properties of TM implementations are *disjoint-access parallelism* and the *invisibility* of read operations. *Disjoint access parallelism* ensures that operations on disconnected data do not interfere, and thus it is critical for TM scalability. The *invisibility* of read operations means that their implementation does not write to the memory, thereby reducing memory contention.

This paper proves an inherent tradeoff for implementations of transactional memories: they cannot be both disjoint-access parallel and have read-only transactions that are invisible and always terminate successfully. In fact, a lower bound of $\Omega(t)$ is proved on the number of writes needed in order to implement a read-only transaction of t items, which successfully terminates in a disjoint-access parallel TM implementation. The results assume *strict serializability* and thus hold under the assumption of *opacity*. It is shown how to extend the results to hold also for weaker consistency conditions, *snapshot isolation* and *serializability*.

Keywords Transactional memory · Disjoint-access parallelism · Partial snapshots · Lower bound · Impossibility result

This research is partially supported by the *Israel Science Foundation* (grant number 953/06) and Intel Corporation. A. Milani is on leave from Sapienza, Universitá di Roma, supported in part by a fellowship from the Lady Davis Foundation and by a grant Progetto FIRB Italia-Israele RBIN047MH9.

H. Attiya (✉) · E. Hillel · A. Milani
Department of Computer Science, Technion, Haifa, Israel
e-mail: hagit@cs.technion.ac.il

E. Hillel
e-mail: eshcar@cs.technion.ac.il

A. Milani
e-mail: milani@labri.fr

1 Introduction

Transactional memory is an attractive paradigm for programming concurrent applications for multicores. A transaction encapsulates a sequence of operations, and it is guaranteed that if any operation takes place, they all do, and that if they do, they appear to other threads to do so atomically, as one indivisible operation. A transactional memory *implementation* translates high-level transaction operations on data items to low-level primitive operations on base objects, containing the data and the meta-data needed for the implementation.

Transactional memory is seriously considered as part of software solutions and as a basis for novel hardware designs. It is therefore imperative to understand inherent tradeoffs in the design and implementation of transactional memory.

One property that is considered critical for the scalability of a transactional memory implementation is *disjoint-access parallelism*: operations on disconnected data should not interfere. Conceptualizing this notion is best done through the *conflict graph* of transactions that overlap in time. Informally, the vertices of the conflict graph represent transactions; an edge connects two *conflicting* transactions, i.e., two transactions that access the same item. Consider, for example, four concurrent transactions: T_1 accessing data items ι_1, ι_2, T_2 accessing data items ι_2, ι_3, T_3 accessing data item ι_3, and T_4 accessing data item ι_4. Figure 1 depicts the conflict graph of the execution interval of these four transactions. This conflict graph contains two edges connecting T_2 with the transactions with which it has conflicts, T_1 and T_3. (See the formal definition in Sect. 2.)

Several transactional memories, e.g. [4, 16], guarantee that transactions access the same base object only if they are connected in the conflict graph. In particular, there is no concurrent access to the shared memory by transactions without common data items. In these implementations, the transactions T_1 and T_4 in the example of Fig. 1, should not access the same base object, since they are not connected in the conflict graph. On the other hand, transactions T_1 and T_3 may access the same base object although they do not have a common data item.

Another important goal is to optimize *read-only transactions*, i.e., transactions that access the memory only through read operations. It is desirable that implementations of read-only transactions do not execute primitive write operations to the memory, so as to reduce memory contention; implementations of read-only transactions that do not write to the memory are called *invisible*. Moreover, since read-only transactions do not write to data items, it seems plausible that they should eventually be able to obtain a consistent view of the data, provided previous versions are kept (as is done in multi-version implementations [24, 26, 27]). Thus, read-only transactions should

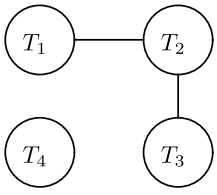

Fig. 1 Example of a simple conflict graph: T_1 accesses ι_1 and ι_2; T_2 accesses ι_2 and ι_3; T_3 accesses ι_3; and T_4 accesses ι_4

(eventually) terminate successfully, regardless of concurrent transactions; such transactions are called *wait-free*.

None of the existing transactional memory implementations is both disjoint-access parallel and has invisible, wait-free read-only transactions. Some are disjoint-access parallel and have invisible but not wait-free read-only transactions [4, 16], while others have invisible, wait-free read-only transactions but are not disjoint-access parallel [26].

Consider, for example, the four transactions above, and assume T_1 is a read-only transaction, while T_2, T_3, and T_4 all write to their data items. The algorithm given in [16], is disjoint-access parallel and has invisible read-only transactions. In some execution, T_1 reads ι_2 then T_1 stops taking steps and T_2 writes to ι_2. The transaction T_2 runs solo without interruption and since the algorithm is obstruction-free T_2 completes successfully. Then, T_1 resumes, reads ι_1, and finally validates its read set at commit time. The value of ι_2 has changed since T_1 read it, and T_1 aborts.

The algorithm given in [26] is a multiversioned transactional memory, with invisible, wait-free read-only transactions, which uses a common counter to totally order committed transactions writing different versions of the same data item. This allows each read-only transaction to return a consistent snapshot of the memory, despite concurrent writing transactions. Due to the common counter, it is not disjoint-access parallel: In the example of Fig. 1, it is possible that all the transactions access the counter, some of them for writing, thus violating disjoint-access parallelism.

This paper shows that there is an inherent tradeoff—no transactional memory implementation can be disjoint-access parallel and have invisible, wait-free read-only transactions—and one of these desirable properties must always be compromised. In fact, we prove a stronger result, showing that in a disjoint-access parallel transactional memory implementation with wait-free read-only transactions, a transaction reading t data items must apply non-trivial primitives (e.g., writes) to at least $t - 1$ base objects. Thus, a read-only transaction must perform one low-level write essentially for each item in its *read set*.

The wait-freedom requirement might seem too restrictive for practical purposes; however, we can prove a similar result where a read-only transaction repeatedly aborts and never terminates successfully; see further discussion in Sect. 6. For read-dominated applications, this implies too much wasted work.

The consistency condition commonly used for transactional memory is *opacity* [10]; very roughly stated, opacity requires all transactions to appear to execute sequentially in an order that agrees with the order of non-overlapping transactions. This is similar to requiring *strict* (*view*) *serializability* [25] applied to all transactions (including each aborted transaction, separately), extended to allow operations other than reads and writes. Our proofs only assume *strict serializability* [25], and hence hold also under the assumption of opacity. In fact, the results also hold for weaker consistency conditions, *snapshot isolation* and *serializability*.

The rest of the paper is organized as follows: Sect. 2 introduces basic definitions and in particular, the notion of disjoint-access parallelism. Section 3.1 presents an impossibility result showing that in a disjoint-access parallel transactional memory implementation with invisible read-only transactions, some read-only transaction may never terminate successfully; this result is proved using only three processes. Section 3.2 strengthens this result and shows that a read-only transaction on t items (in

a disjoint-access parallel transactional memory implementation with wait-free read-only transactions) must apply write primitives to $t-1$ base objects; this result requires $t+1$ processes. Section 4 extends the results to hold even with the weaker conditions of snapshot isolation and serializability. We discuss related work in Sect. 5, and conclude in Sect. 6.

2 Preliminaries

A *transaction* is a sequence of operations executed by a single process on a set of *data items* shared with other transactions; all data items are initially 0. We assume data items are accessed by simple *read* and *write* operations; our impossibility results clearly hold for transactional memory that also supports other operations. A complete interface of transactional memory also includes *commit* and *abort* operations, which we do not model here, since they are not needed for our impossibility results.

The collection of data items accessed by a transaction is the transaction's *data set*; in particular, the items written by the transaction are its *write set*, and the items read by the transaction are its *read set*. A transaction whose *write set* is empty, is said to be a *read-only transaction*. We assume the transaction's read set and write set are provided at the start of the transaction, and do not elaborate further on the manner a transaction issues its operations; this only makes our impossibility results stronger.

An *implementation* of *software transactional memory* (abbreviated *STM*) provides data representation for transactions and data items using *base objects*, and algorithms, specified as *primitive operations* (abbreviated *primitives*) on the base objects, which *asynchronous* processes have to follow in order to execute the operations of transactions. In addition to ordinary read and write (low level) primitives, we allow arbitrary read-modify-write primitives, like `CAS`, even those accessing several locations simultaneously.

A primitive is *non-trivial* if it may change the value of the object, e.g., a write or `CAS`; otherwise, it is *trivial*, e.g., a read.

An *event* is a computation *step* by a process consisting of local computation and the application of a primitive to base objects, followed by a change to the process's state, according to the results of the primitive. A *configuration* is a complete description of the system at some point in time, i.e., the state of each process and the value of each shared base object. There is a unique *initial* configuration in which every process is in its initial state and every base object contains its initial value.

An *execution interval* α is a finite or infinite alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \ldots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1}, for every $k = 0, 1, \ldots$. An *execution* is an execution interval in which C_0 is the initial configuration.

Two executions α_1 and α_2 are *indistinguishable* to a process p, if p goes through the same sequence of state changes in α_1 and in α_2; in particular, p executes the same sequence of computation steps, which return the same results.

We point out that the model encompasses two levels of abstraction: The high level has transactions, each of which is a sequence of operations accessing data items. At the low level, these transactions are translated into executions in which a sequence of

events apply primitive operations to base objects, containing the data and the meta-data needed for the implementation.

2.1 STM Properties

The *interval of a transaction* T is the execution interval that starts at the first event of T and ends at the last event of T, if there is one, taken by the process executing the algorithm for T. If T does not have a last event in the execution, then the interval of T is the (possibly infinite) execution interval starting at the first event of T. Two transactions *overlap* if their intervals overlap. A configuration C is *quiescent* if no transaction is pending in C, i.e., it is not inside the interval of any transaction.

An STM is *serializable* if transactions appear to execute sequentially, one after the other [25]; we assume that this *serialization order* preserves the *per-process order*, i.e., transactions of the same process maintain their order. Since infinite executions also have to be serializable, it follows that if transactions by the same process read a data item, they eventually return the last value written to it. Traditional definitions of serializability (e.g., [25, 29]) apply only to finite executions, and hence, admit trivial implementations, where read operations always return the initial values of the data items.

An STM is *strictly serializable* if the serialization order preserves the order of non-overlapping transactions [25]; this notion is called *order-preserving serializability* in [29], and is the analogue of *linearizability* [15] for transactions. Note that strict serializability is implied by the *opacity* correctness condition, recently defined for transactional memory [10].

We assume that a transaction terminates successfully if it runs alone from a quiescent configuration. This property is satisfied by *obstruction-free* STM implementations, in which a process that eventually runs alone for long enough makes progress, i.e., transactions terminate successfully when eventually executing solo [16]. This property is also satisfied by STM implementations that are *weakly progressive* [11], in which a transaction that does not encounter conflicts has to terminate successfully; note that blocking, lock-based STM implementations like TL2 [7] are weakly progressive.

2.2 Memory disjoint-access parallelism

An important property STM implementations have to provide is allowing unrelated transactions to progress independently, even if they are concurrent. Below, we formally define what it means for two transactions to be *unrelated* through a conflict graph that represents the relations between transactions. Then we define *disjoint-access parallelism*, a property that captures the intuition that an implementation should not cause two transactions, which are unrelated at the high-level, to simultaneously access the same low-level shared memory.

The *conflict graph* of an execution interval I is an undirected graph, where vertices represent transactions; an edge connects two transactions that access the same item. Two transactions T_1 and T_2 are *disjoint-access* if there is no path between them in the conflict graph of the minimal execution interval containing the intervals of T_1 and T_2.

Two events *contend* on a base object o if they both access o, and at least one of them applies a non-trivial primitive to o. Two processes *concurrently contend* on a base object o if they have pending events at the same configuration that contend on o.

Definition 1 An STM implementation is *weakly disjoint-access parallel* if two processes p_1 and p_2, executing transactions T_1 and T_2, concurrently contend on the same base object, only if T_1 and T_2 are not disjoint-access.

This definition captures the first condition of the disjoint-access parallelism property of Israeli and Rappoport [20], in accordance with most of the literature (cf. [14]). Our requirement is weaker than theirs, as we allow two processes to apply a trivial primitive on the same base object, e.g., read, when executing two transactions even if they are disjoint-access. Moreover, our definition only prohibits concurrent contending accesses, allowing transactions to contend on a base object o at different points of the execution; we shall see in Lemma 2 that, under some conditions, these transactions can be made to concurrently contend on o.

The original definition [20] also restricts the impact of concurrent transactions on the *step complexity* of a transaction; our results do not rely on this additional condition, making them stronger.

3 Strictly Serializable STMs

3.1 Impossibility of Invisible Read-Only Transactions

A read-only transaction is *invisible* if its algorithm only applies trivial primitives to base objects. We prove that in a disjoint-access parallel STM implementation with invisible read-only transactions, some read-only transaction will not terminate successfully in a finite number of steps; this is formally stated in Theorem 1.

Specifically, we construct an infinite execution of a read-only transaction. This execution consists of a single read-only transaction with one complete update transaction between any pair of consecutive steps by the read-only transaction; an *update* is a transaction with a singleton write set and an empty read set. We first define a special (finite) execution of this form, called *flippable*, and show that such a read-only transaction cannot terminate successfully. Then we show how a flippable execution can be repeatedly extended to construct successively longer flippable executions.

An execution is called flippable since there are two similar executions in which we flip the position of two update transactions and one of the executions is indistinguishable from the original execution. One type of flipped execution is called a *forward* flip since an update transaction is moved earlier in the execution, while other is called a *backward* flip since an update transaction is deferred in the execution. Formally:

Definition 2 A *flippable execution of length k with t updaters* is a finite execution $E_k = U_0 s_1 U_1 \ldots s_k U_k$ executed by processes $p_0, \ldots, p_{t-1}$ executing update transactions and process q executing a read-only transaction, which reads and returns the value of t data items $t_0 \ldots t_{t-1}$. The execution E_k satisfies all the following conditions:

1. for $j = 1, \ldots, k$, s_j is a single step by q,
2. for $j = 0, \ldots, k$, U_j is a solo execution of a complete update transaction, in which process $p_h \in \{p_0, \ldots, p_{t-1}\}$, writes $j+1$ to the data item ι_h,
3. consecutive updates are executed by different processes, and
4. for any l, $0 < l \leq k$, the execution

$$E_k = U_0 s_1 U_1 \ldots s_{l-1} U_{l-1} s_l U_l \ldots s_k U_k$$

is indistinguishable to all processes from one of the following executions:

$$\overleftarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} U_l U_{l-1} s_l \ldots s_k U_k$$

in which the update transaction U_l is executed before $U_{l-1} s_l$ instead of after $U_{l-1} s_l$ (*forward flip*) or

$$\overrightarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} s_l U_l U_{l-1} \ldots s_k U_k$$

in which the update transaction U_{l-1} is executed after $s_l U_l$ instead of before $s_l U_l$ (*backward flip*).

Figures 2(b) and 2(c) present the forward and the backward flips of the execution in Fig. 2(a).

This definition, and the structure of our proof, is similar to the lower bound of Attiya, Ellen and Fatourou [2] on the step complexity of update operations in implementations of atomic snapshot objects. The main difference is that our definition of a flippable execution has *two* types of flipped executions, and t processes executing update transactions instead of just two.

The next lemma proves that if the implementation has a flippable execution then the read-only transaction in this execution does not terminate; it is proved by arguments similar to those applied in [2], extended to handle the possibility of two kinds of flips (forward and backward).

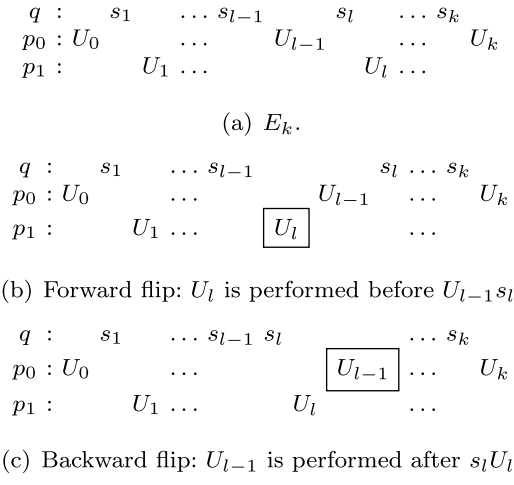

Fig. 2 A flippable execution of length k with two updaters: (**a**) shows a flippable execution E_k; (**b**) shows the *forward* flip execution of E_k, where the update transaction U_l by process p_1 is executed before the update transaction U_{l-1} by process p_0 and before the step s_l of the read-only transaction; (**c**) shows the *backward* flip execution of E_k, where the update U_{l-1} by process p_0 is deferred after the update transaction U_l by process p_1 and after the step s_l of the read-only transaction

Lemma 1 *The read-only transaction in a flippable execution does not terminate successfully.*

Proof Let $E_k = U_0 s_1 U_1 \ldots s_k U_k$ be a flippable execution. Assume, towards a contradiction, that q successfully terminates its read-only transaction in E_k, with a result $(v_0, \ldots, v_{t-1})$. The proof first fixes the serialization of the update transactions, and then shows that it is not possible to serialize the read-only transaction among the update transactions, using the forward and backward flip executions, which are indistinguishable to q from E_k.

Since the update transactions in the execution E_k do not overlap, they must be serialized in the order $U_0, \ldots, U_k$. Since all steps of the read-only transaction by q are after U_0 and before U_k, it has a unique serialization point between U_{l-1} and U_l, for some l, $1 \leq l \leq k$. Let ι_h be the item written by U_{l-1}, and recall that U_{l-1} writes l to ι_h; hence $v_h = l$.

The execution E_k is indistinguishable to process q from F_l, which is either the forward flip

$$\overleftarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} U_l U_{l-1} s_l s_{l+1} \ldots U_k$$

in which update U_l is executed before $U_{l-1} s_l$ instead of after $U_{l-1} s_l$; or the backward flip

$$\overrightarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} s_l U_l U_{l-1} s_{l+1} \ldots U_k$$

in which update U_{l-1} is executed after $s_l U_l$ instead of before $s_l U_l$. Hence, the read-only transaction executed by q in F_l returns the same vector, $(v_0, \ldots, v_{t-1})$, as in E_k.

Since the update transactions do not overlap in F_l, they are serialized in the order $U_0, \ldots, U_l, U_{l-1}, \ldots, U_k$, that is, the same as for E_k, except that U_{l-1} and U_l are flipped. Since two consecutive update transactions are to different items, the values of $\{\iota_0, \ldots, \iota_{t-1}\}$ are the same after both update transactions have been executed, no matter which has been executed first. Hence, at all points in the serialization of F_l, except between U_l and U_{l-1}, the value of all items $\{\iota_0, \ldots, \iota_{t-1}\}$ is the same as its value in the corresponding points in the serialization of E_k. Thus, the read-only transaction of q can only be serialized after U_l and before U_{l-1} in F_l. However, since U_{l-1} is the first write of l to ι_h, the value of ι_h is not l before U_{l-1}, and hence, the read-only transaction executed by q cannot be serialized between U_l and U_{l-1}. This contradicts the assumption that the read-only transaction terminates successfully. □

It remains to prove that a flippable execution exists. Lemma 3 (below) shows how to inductively construct a flippable execution, when read-only transactions are invisible. The crux of this lemma is quite different from [2], as it relies on weakly disjoint-access parallelism. A critical step in the proof is provided by Lemma 2, showing that in a weakly disjoint-access parallel STM, two consecutive updates by different processes on different items cannot contend on the same base objects. Note that two consecutive update transactions do not contradict weak disjoint-access parallelism since the steps of their executing processes are not interleaved. The proof of the next lemma shows that two such consecutive updates can be perturbed to *concurrently* contend on the same base object.

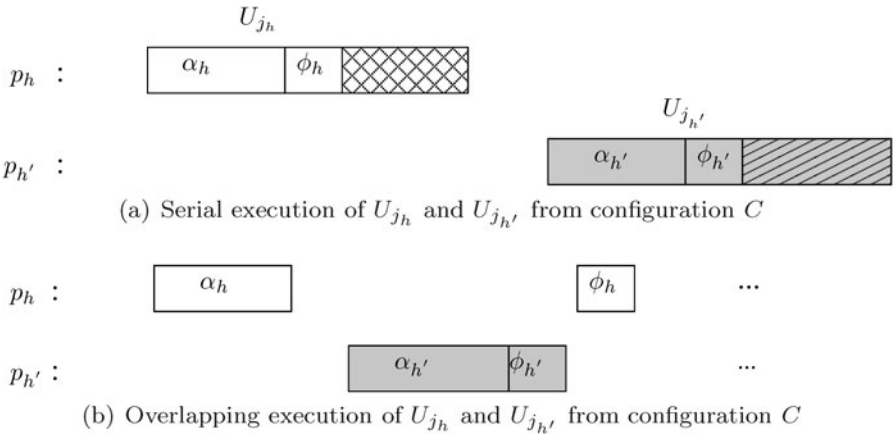

Fig. 3 Illustration for the proof of Lemma 2

Lemma 2 *Given a weakly disjoint-access parallel STM implementation and a quiescent configuration C, consider the consecutive execution of two update transactions $U_{j_h}U_{j_{h'}}$, executed by a process p_h on an item ι_h and by process $p_{h'}$ on an item $\iota_{h'}$, $h \neq h'$, respectively, from C. Then p_h and $p_{h'}$ do not contend on the same base object when executing U_{j_h} and $U_{j_{h'}}$.*

Proof Assume, towards a contradiction, that p_h and $p_{h'}$ contend on a base object when executing $U_{j_h}U_{j_{h'}}$ from a quiescent configuration C. If in U_{j_h}, p_h applies a non-trivial primitive to a base object on which they contend, let ϕ_h be the last event in U_{j_h} in which p_h applies such a primitive, say, to base object o. Let $\phi_{h'}$ be the first event in $U_{j_{h'}}$ that accesses o.

Otherwise, p_h only applies trivial primitives in U_{j_h} to base objects on which it contends with $p_{h'}$ in $U_{j_{h'}}$; let $\phi_{h'}$ be the first event in $U_{j_{h'}}$ in which $p_{h'}$ applies a non-trivial primitive to some base object, say, o, on which they contend. Let ϕ_h be the last event of p_h in U_{j_h} that accesses o.

In both cases, denote by $\alpha_h\phi_h$ the prefix of the execution of U_h from C and by $\alpha_{h'}\phi_{h'}$ the prefix of the execution of $U_{h'}$ after U_h (see Fig. 3(a)).

We now create an overlapping execution of the update transactions U_{j_h} and $U_{j_{h'}}$, by processes p_h and $p_{h'}$, from C. We argue that p_h and $p_{h'}$ perform the same steps up to the events ϕ_h and $\phi_{h'}$, and as illustrated in Fig. 3(b), p_h and $p_{h'}$ concurrently contend on base object o.

In more detail, consider the execution $\alpha_h\alpha_{h'}$ from C, in which p_h executes U_{j_h} until it is about to perform ϕ_h, and then $p_{h'}$ executes $U_{j_{h'}}$ until it is about to perform $\phi_{h'}$. Clearly, p_h is about to perform ϕ_h also after $\alpha_h\alpha_{h'}$. By construction, the execution interval $\alpha_h\alpha_{h'}$ from C is indistinguishable to $p_{h'}$ from the execution interval $U_{j_h}\alpha_{h'}$ from C. Hence, $p_{h'}$ is about to perform the event $\phi_{h'}$ also after $\alpha_h\alpha_{h'}$, that is, $p_{h'}$ and p_h concurrently contend on o. However, the conflict graph of the execution interval $\alpha_h\alpha_{h'}\phi_{h'}\phi_h$ does not contain a path between the data sets of U_{j_h} and $U_{j_{h'}}$, contradicting the assumption that the implementation is weakly disjoint-access parallel. □

Since two consecutive updates do not contend on the same base object, we can construct an execution where either the previous update is deferred or the next update is moved forward in the execution without affecting the single step of the read-only transaction in between them. This allows us to inductively construct a flippable execution, in the proof of the next lemma.

Lemma 3 *For every $k \geq 0$, every weakly disjoint-access parallel implementation of an STM with invisible read-only transactions, has a flippable execution $E_k = U_0 s_1 U_1 s_2 \ldots U_k$ with two updaters p_0 and p_1, which is indistinguishable to p_0 and p_1 from the execution $E'_k = U_0 U_1 \ldots U_k$ in which only p_0 and p_1 take steps.*

Proof The proof is by induction on the length, k, of the flippable execution E_k executed by a process q and two updaters p_0 and p_1 on two items $\{\iota_0, \iota_1\}$. In the base case, $k = 0$, the lemma holds with a solo execution of U_0, an update transaction by p_0 that writes 1 to ι_0. U_0 successfully terminates since it runs solo from a quiescent configuration.

For the induction step, consider a flippable execution of length $k \geq 1$, $E_k = U_0 s_1 U_1 s_2 \ldots U_k$, which is indistinguishable to p_0 and p_1 from the execution $E'_k = U_0 U_1 \ldots U_k$. We show how to construct a flippable execution of length $k+1$, which is indistinguishable from an execution in which only p_0 and p_1 take steps.

By Lemma 1, the read-only transaction does not terminate successfully in E_k. Let s_{k+1} be the next step by q. Assume U_k is executed by $p_{h'}$ and let $h = 1 - h'$; note that $h \neq h'$. Let $E_{k+1} = E_k s_{k+1} U_{k+1}$, where process p_h writes $k+2$ to ι_h in the update transaction U_{k+1}. Note that U_{k+1} terminates successfully: The configuration at the end of $E_{k+1} = E_k s_{k+1}$ is indistinguishable from the configuration at the end of E'_k, which is quiescent; since the execution of U_{k+1} from the configuration at the end of E'_k must terminate successfully, by our progress condition, U_{k+1} must also terminate successfully when executing from the configuration at the end of $E_{k+1} = E_k s_{k+1}$.

Since the read-only transaction by q is invisible, $E_{k+1} U_{k+1}$ is indistinguishable to p_0 and p_1 from the execution $E'_k U_{k+1}$.

It remains to prove that E_{k+1} is a flippable execution, i.e., that for every l, $0 < l \leq k+1$, the execution E_{k+1} is indistinguishable to all processes from either $\overleftarrow{F}_l$ or $\overrightarrow{F}_l$. For every l, $0 < l \leq k$, by the inductive assumption, the execution

$$E_k = U_0 s_1 U_1 \ldots s_{l-1} U_{l-1} s_l U_l \ldots s_k U_k$$

is indistinguishable to all processes from the flipped execution F_l which is either

$$\overleftarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} U_l U_{l-1} s_l \ldots U_k$$

or

$$\overrightarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} s_l U_l U_{l-1} \ldots s_k U_k.$$

In particular, the configurations at the end of the two executions E_k and F_l are the same. Hence, $E_{k+1} = E_k s_{k+1} U_{k+1}$ and $F_l s_{k+1} U_{k+1}$ are indistinguishable to all processes.

To prove the condition for $l = k+1$, let C'_{k-1} be the configuration at the end of E'_{k-1}; C'_{k-1} is quiescent, and Lemma 2 implies that $p_{h'}$ and p_h do not contend on the same base object when executing U_k followed by U_{k+1} from C'_{k-1}, namely, in the suffix of E'_{k+1}. By the indistinguishability of E'_{k+1} and E_{k+1}, $p_{h'}$ and p_h do not contend on the same base object while executing U_k and U_{k+1} also in the execution E_{k+1}. Moreover, if q accesses a base object o in s_{k+1}, then either at least one of the two processes p_h or $p_{h'}$ does not access o in U_{k+1} or U_k, respectively, or they both apply a trivial primitive to o. In the former case, if p_h does not access o in U_{k+1} then

$$\overleftarrow{F}_{k+1} = U_0 s_1 U_1 \ldots s_k U_{k+1} U_k s_{k+1}$$

is indistinguishable to all processes from E_{k+1}, while if $p_{h'}$ does not access o in U_k, then

$$\overrightarrow{F}_{k+1} = U_0 s_1 U_1 \ldots s_k s_{k+1} U_{k+1} U_k$$

is indistinguishable to all processes from E_{k+1}. If both p_h and $p_{h'}$ apply a trivial primitive to o, then both flipped executions, $\overleftarrow{F}_{k+1}$ and $\overrightarrow{F}_{k+1}$, are indistinguishable to all processes from E_{k+1}. □

The impossibility result follows from Lemmas 1 and 3.

Theorem 1 *There is no weakly disjoint-access parallel implementation with invisible read-only transactions of a strictly serializable STM, in which read-only transactions always terminate successfully.*

The impossibility result stated in Theorem 1 holds also for opaque STMs [10], since opacity implies strict serializability.

3.2 Lower Bound for Read-Only Transactions

The technique of the previous section can be extended to prove that a read-only transaction of t items in a disjoint-access parallel STM implementation, which successfully terminates in a finite number of steps, must apply non-trivial primitives to $t-1$ base objects; this assumes that there are at least $t+1$ processes.

The proof of Lemma 1—showing that the read-only transaction in a flippable execution cannot terminate successfully—does not rely on the fact that the read-only transaction is invisible, and the lemma continues to hold. On the other hand, we must modify the proof showing the existence of the flippable execution.

This result relies on a stronger notion of disjoint-access parallelism, which requires two transactions to be connected (in the conflict graph) even if they both just apply a trivial primitive to the same base object. (This is the definition in [20].) Two processes *concurrently access* a base object o if both have a pending access to o at some configuration.

Definition 3 An STM implementation is *disjoint-access parallel* if two processes p_1, p_2 concurrently access the same base object when executing transactions T_1 and T_2, respectively, only if T_1 and T_2 are not disjoint-access.

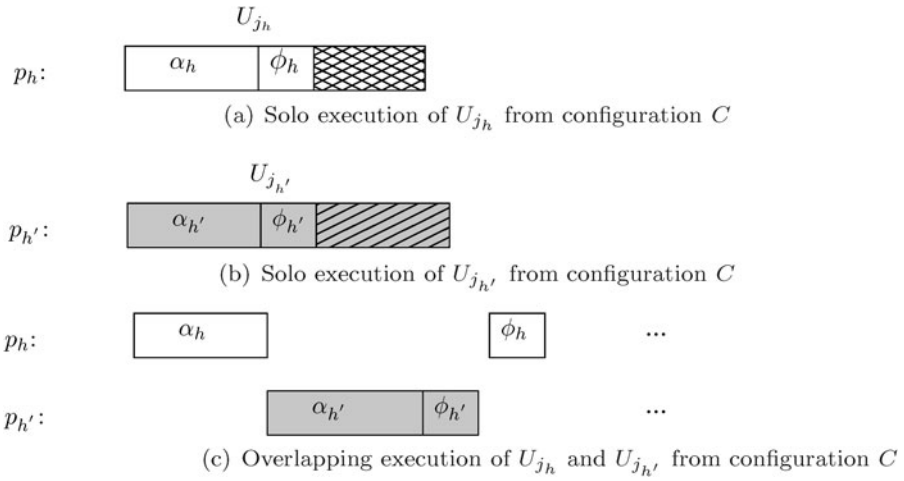

Fig. 4 Illustration for the proof of Lemma 4

Since we now put a stronger requirement on disjoint-access parallel STM implementations, Lemma 2, assuming a weaker requirement, still holds.

We first show (in Lemma 4) that, in a disjoint-access parallel STM implementation, two update transactions executed by different processes on different items do not access a common base object when each of them runs solo from a quiescent configuration. This is used in Lemma 5 to prove the existence of a flippable execution, when a read-only transaction of t data items applies non-trivial primitives to at most $t-2$ base objects.

Lemma 4 *Given a disjoint-access parallel STM implementation and a quiescent configuration C, consider the execution of an update transaction U_{j_h} to the item ι_h by process p_h, and an update transaction $U_{j_{h'}}$ to the item $\iota_{h'}$ by process $p_{h'}$, $h \neq h'$, from C. Then, p_h and $p_{h'}$ do not access a common base object when executing U_{j_h} and $U_{j_{h'}}$, respectively.*

Proof Assume, towards a contradiction, that p_h and $p_{h'}$ access the same base object while executing U_{j_h} and $U_{j_{h'}}$, respectively, from C. Let o be the first base object accessed by p_h that is also accessed by $p_{h'}$. Let $\alpha_h\phi_h$ be the prefix of the execution of U_{j_h} from C, where ϕ_h is the first event in which p_h accesses o (see Fig. 4(a)). Let $\alpha_{h'}\phi_{h'}$ be the prefix of the execution of $U_{j_{h'}}$ from C, where $\phi_{h'}$ is the first access of $p_{h'}$ to o (see Fig. 4(b)). We show how to paste the executions so that the events ϕ_h and $\phi_{h'}$ are concurrently pending.

Consider the execution $\alpha_h\alpha_{h'}$ from C, where p_h executes U_{j_h} until it is about to access o, and then $p_{h'}$ executes $U_{j_{h'}}$ until it is about to access o (see Fig. 4(c)). By construction, the execution $\alpha_h\alpha_{h'}$ from C is indistinguishable to p_h and $p_{h'}$ from the corresponding executions α_h and $\alpha_{h'}$ from C. Thus, $p_{h'}$ has the event $\phi_{h'}$ pending and p_h has the event ϕ_h pending after $\alpha_h\alpha_{h'}$; that is, $p_{h'}$ and p_h concurrently access o. However, in the conflict graph of the execution interval $\alpha_h\alpha_{h'}\phi_{h'}\phi_h$ from C, there is

no path between the data sets of U_{j_h} and $U_{j_{h'}}$, contradicting the assumption that the implementation is disjoint-access parallel. □

We show that at any point during the execution of the read-only transaction, there is a process that can write to its item without accessing any base object to which q applies non-trivial primitives, thus making the read-only transaction "invisible" to the other processes. Note that, by the definition of a flippable execution, each process always updates the same item. We prove such a process exists by applying a "pigeon hole" argument to show that the process does not access any base object to which the read-only transaction applies non-trivial primitives. Since there are $t-1$ processes to choose from, each accessing a different item, and since the read-only transaction applies non-trivial primitives to at most $t-2$ base objects, at least two update transactions by different processes access the same base object, which can be shown to violate disjoint-access parallelism.

Lemma 5 *For every $k \geq 0$, a disjoint-access parallel implementation of an STM in which a read-only transaction of $t > 2$ data items applies non-trivial primitives to at most $t-2$ base objects, has a flippable execution $E_k = U_0 s_1 U_1 s_2 \ldots U_k$ with t updaters, which is indistinguishable to $p_0, \ldots, p_{t-1}$ from the execution $E'_k = U_0 U_1 \ldots U_k$ in which only $p_0, \ldots, p_{t-1}$ take steps.*

Proof The proof is by induction on the length k of the flippable execution E_k. The base case is when $k = 0$. The lemma holds with a solo execution of an update transaction, U_0, by process p_0 that writes 1 to ι_1. U_0 successfully terminates since it runs solo from a quiescent configuration.

For the induction step, consider a flippable execution of length k, $E_k = U_0 s_1 U_1 s_2 \ldots U_k$, which is indistinguishable to $p_0, \ldots, p_{t-1}$ from the execution $E'_k = U_0 U_1 \ldots U_k$. We show how to construct a flippable execution of length $k+1$, which is indistinguishable to $p_0, \ldots, p_{t-1}$ from an execution in which only $p_0, \ldots, p_{t-1}$ take steps.

By Lemma 1, the read-only transaction does not terminate successfully in E_k. Let s_{k+1} be the next step by q and let C_{k+1} denote the configuration at the end of $E_k s_{k+1}$; also, let C'_{k+1} be the configuration at the end of E'_k.

The process p_h to execute U_{k+1} is chosen from $p_0, \ldots, p_{t-1}$ such that p_h did not execute U_k and a solo execution of U_{k+1} from C_{k+1} by p_h does not access any base objects to which q applies non-trivial primitives in $E_k s_{k+1}$. Note that this transaction must terminate successfully, by our progress condition; although C_{k+1} is not quiescent, it is indistinguishable from C'_{k+1}, which is quiescent.

We claim such a process exists. Assume, towards a contradiction, that for every process $p_{h_{k+1}}$, $h_{k+1} \neq h_k$, the solo execution by $p_{h_{k+1}}$ from C_{k+1} of the update transaction that writes $k+2$ to $\iota_{h_{k+1}}$ accesses a base object to which q applies a non-trivial primitive in $E_k s_{k+1}$. We consider $t-1$ possible processes, each writing to a different item. Since the read-only transaction applies non-trivial primitives to at most $t-2$ base objects, at least two update transactions executed by different processes p_h and $p_{h'}$ to different items ι_h and $\iota_{h'}$, starting from configuration C_{k+1}, access the same base object in their first access to a base object to which q applies a non-trivial primitive. Recall that C'_{k+1} is quiescent. Since the execution $E_k s_{k+1}$ is indistinguishable

to processes p_h and $p_{h'}$ from the execution E'_k, they access the same base object also when executing the update transactions from C'_{k+1}, which by Lemma 4, violates the assumption that the implementation is disjoint-access parallel.

Pick some process $p_{h_{k+1}}$, $h_{k+1} \neq h_k$, that does not access any base objects to which q applies non-trivial primitives in $E_k s_{k+1}$; let U_{k+1} be an update by $p_{h_{k+1}}$ that writes $k+2$ to $\iota_{h_{k+1}}$ and denote $E_{k+1} = E_k s_{k+1} U_{k+1}$.

Next, we prove that the execution E_{k+1} is indistinguishable to $p_0, \ldots, p_{t-1}$ from the execution E'_{k+1}. This holds for processes other than $p_{h_{k+1}}$ by the inductive assumption and since these processes take no steps in the suffix of this execution. For $p_{h_{k+1}}$, this holds by the inductive assumption and since the solo execution U_{k+1} of an update transaction by $p_{h_{k+1}}$ does not access base objects to which q applies a non-trivial primitive in $E_k s_{k+1}$.

It remains to prove that for every l, $0 < l \leq k+1$, the execution E_{k+1} is indistinguishable to all processes from the flipped execution F_l which is either $\overleftarrow{F}_l$ or $\overrightarrow{F}_l$, as defined in Definition 2. For every l, $0 < l \leq k$, by the inductive assumption, the execution

$$E_k = U_0 s_1 U_1 \ldots s_{l-1} U_{l-1} s_l U_l \ldots s_k U_k$$

is indistinguishable to all processes from the flipped execution F_l which is either

$$\overleftarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} U_l U_{l-1} s_l \ldots U_k$$

or

$$\overrightarrow{F}_l = U_0 s_1 U_1 \ldots s_{l-1} s_l U_l U_{l-1} \ldots s_k U_k.$$

In particular, the configurations at the end of the two executions E_k and F_l are the same. Hence, the executions $E_{k+1} = E_k s_{k+1} U_{k+1}$ and $F_l s_{k+1} U_{k+1}$ are indistinguishable to all processes.

For $l = k+1$, consider the flipped executions $\overleftarrow{F}_{k+1}$ and $\overrightarrow{F}_{k+1}$. The configuration C'_{k-1} at the end of E'_{k-1} is quiescent. Any STM implementation which is disjoint-access parallel is also weakly disjoint-access parallel, hence we can apply Lemma 2 to deduce that p_{h_k} and $p_{h_{k+1}}$ do not contend on, and hence do not access the same base object while executing U_k and U_{k+1} from C'_{k-1}. The indistinguishability property implies that p_{h_k} and $p_{h_{k+1}}$ do not access the same base object while executing U_k and U_{k+1} also in E_{k+1}.

Moreover, if q applies a trivial primitive to some base object o in s_{k+1}, then either at least one of the two processes $p_{h_{k+1}}$ and p_{h_k} does not access o in U_{k+1} and in U_k respectively, or they both apply a trivial primitive to o. In the former case, if $p_{h_{k+1}}$ does not access in U_{k+1} any object that q accesses in s_{k+1}, then

$$\overleftarrow{E}_{k+1} = U_0 s_1 U_1 \ldots s_k U_{k+1} U_k s_{k+1}$$

is indistinguishable to all processes from E_{k+1}, while if p_{h_k} does not access in U_k any object that q accesses in s_{k+1}, then

$$\overrightarrow{E}_{k+1} = U_0 s_1 U_1 \ldots s_k s_{k+1} U_{k+1} U_k$$

is indistinguishable to all processes from E_{k+1}. If $p_{h_{k+1}}$ and p_{h_k} apply a trivial primitive to o, then both flipped executions are indistinguishable to all processes from E_{k+1}. □

The lower bound follows:

Theorem 2 *In a strict serializable disjoint-access parallel STM implementation for $t+1$ processes, where all read-only transactions terminate successfully, some read-only transaction of $t > 2$ data items applies non-trivial primitives to at least $t-1$ base objects.*

This lower bound holds also for opaque STMs, since opacity implies strict serializability.

4 Extending the Results to Weaker Consistency Conditions

In this section, we show that both Theorems 1 and 2 hold for weaker consistency conditions, namely, snapshot isolation and serializability.

4.1 Snapshot Isolation

Snapshot isolation [6, 23, 27, 29] decouples the consistency of the reads and the writes. Informally, all read operations in a transaction return the value of the most recent value as of the time the transaction starts. In addition, the write sets of any pair of concurrent transactions must be disjoint. For a formal definition, see [29, Definition 10.3].

We prove an analogue of Lemma 1, that is, we show that the read-only transaction by process q in a flippable execution cannot terminate successfully, also when the implementation provides snapshot isolation.

Lemma 6 *Consider a flippable execution of length $k \geq 0$ with t updaters, $E_k = U_0 s_1 U_1 \ldots s_k U_k$, of an STM that provides snapshot isolation. The read-only transaction by process q does not terminate successfully.*

Proof Assume, towards a contradiction, that q successfully terminates its read-only transaction in E_k, with a result $(v_0, \ldots, v_{t-1})$. Let ι_{f_l} be the item written by U_l; recall that the initial value of all items is zero and that U_l writes $l+1$ to ι_{f_l}. By the definition of snapshot isolation, each read operation from an item in the read-only transaction by q returns the most recent committed write operation that updated this item as of the time the read-only transaction starts. The read-only transaction by q returns the most recent values after U_0 is executed and before any other update is executed. Hence, $v_{f_0} = 1$, and for every l, $1 \leq l \leq t-1$, $v_{f_l} = 0$.

The execution E_k is indistinguishable to process q from F_1, which is either the forward flip

$$\overleftarrow{F}_1 = U_1 U_0 s_1 s_2 U_2 \ldots U_k$$

or the backward flip

$$\overrightarrow{F}_1 = s_1 U_1 U_0 s_2 U_2 \ldots U_k.$$

Hence, the read-only transaction executed by q in F_1 returns the same vector as in E_1.

However, the definition of snapshot isolation requires different results for the read-only transaction in these executions. In $\overleftarrow{F}_1$, the read-only transaction returns the most recent values after U_0 and U_1 are executed and before any other update is executed, hence it returns a vector where $v_{f_0} = 1$, $v_{f_1} = 2$, and $v_{f_l} = 0$, for every l, $2 \leq l \leq t-1$. In $\overrightarrow{F}_1$, the read-only transaction returns the most recent values before any update is executed, hence it returns a vector where $v_{f_l} = 0$, for every l, $0 \leq l \leq t-1$. Thus, the read-only transaction cannot terminate successfully. □

Arguments similar to Sect. 3 can be used to derive the following impossibility result and lower bound:

Theorem 3 *There is no weakly disjoint-access parallel STM implementation with invisible read-only transactions of an STM providing snapshot isolation, in which read-only transactions always terminate successfully.*

Theorem 4 *In a disjoint-access parallel STM implementation for $t+1$ processes providing snapshot isolation, where all read-only transactions terminate successfully, some read-only transaction of $t > 2$ data items applies non-trivial primitives to at least $t-1$ base objects.*

4.2 Serializability

Recall that an STM is *serializable* if transactions appear to execute sequentially, one after the other; note that we require that transactions of the same process preserve their order (*per-process* order) and that repeatedly reading the same data item eventually returns a non-initial value.

The proof uses an additional process q'. Given a flippable execution $E_k = U_0 s_1 U_1 \ldots s_k U_k$, we construct an *augmented flippable execution*

$$\widehat{E}_k = U_0 s_1 S_1^* U_1 \ldots s_k S_k^* U_k,$$

where the additional process q' performs invisible read-only transactions. For every $j \in \{1, \ldots, k\}$, q' performs solo a sequence S_j^* of read-only transactions after the event s_j by process q and before the update U_j. Each read-only transaction in S_j^* accesses the items $\iota_{f_{j-1}}$ and ι_{f_j} updated by U_{j-1} and U_j. The result of the last read-only transaction in the sequence S_j^*, denoted S_j, is the value written by U_{j-1} to $\iota_{f_{j-1}}$ and the last value of ι_{f_j} before U_j updates it.

Figure 5 shows the augmented flippable execution obtained by augmenting the flippable execution E_k of Fig. 2 with sequences of read-only transactions performed by process q'.

We apply the per-process ordering of transactions to prove that the read-only transactions of q' must eventually read the latest value written in U_{j-1}, and thus, S_j^* is finite.

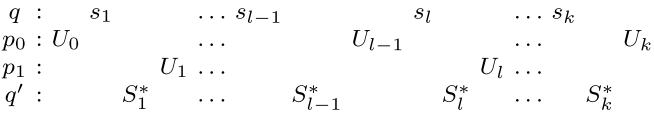

Fig. 5 An augmented flippable execution $\widehat{E}_k$ derived from the flippable execution E_k of Fig. 2

Lemma 7 *Consider an augmented flippable execution of length $k \geq 0$, $\widehat{E}_k = U_0 s_1 S_1^* U_1 \ldots s_k S_k^* U_k$. In any serialization of $\widehat{E}_k$ that preserves the per-process order, $U_0, U_1, \ldots, U_k$ appear in their order of execution.*

Proof We show, by induction on ℓ, that $U_0, U_1, \ldots, U_\ell$ appear in their order of execution. In the base case, $k = 0$, the serialization of U_0 is trivial.

For the induction step, consider $U_{\ell+1}$. By the induction assumption, the updates $U_0, U_1, \ldots, U_\ell$ are serialized by their execution order in $\widehat{E}_k$. By construction, $S^*_{\ell+1}$ is a sequence of read-only transactions that access ι_{f_ℓ} and $\iota_{f_{\ell+1}}$, and the last read-only transaction in $S^*_{\ell+1}$, denoted $S_{\ell+1}$, returns the value written by U_ℓ and the last value of $\iota_{f_{\ell+1}}$ before the one written by $U_{\ell+1}$.

The sequence $S^*_{\ell+1}$ is finite since the STM is serializable and so, eventually, some transaction must return the latest values written to ι_{f_ℓ} and $\iota_{f_{\ell+1}}$, and by the induction assumption, U_ℓ is the last to write to ι_{f_ℓ}. Moreover, $S_{\ell+1}$ completes before $U_{\ell+1}$ starts, so it cannot return the value written by $U_{\ell+1}$, since due to serializability, a read operation can not return a value not written.

Since each data item is written by a different process, and due to per-process order, $U_{\ell+1}$ can not be serialized before the last update of $\iota_{f_{\ell+1}}$ preceding $U_{\ell+1}$.

Moreover, $U_{\ell+1}$ can not be serialized after this update and before $S_{\ell+1}$, since $S_{\ell+1}$ does not return the value written by $U_{\ell+1}$. Hence, $U_{\ell+1}$ is serialized after $S_{\ell+1}$. □

We use Lemma 7 to prove an analogue of Lemma 1.

Lemma 8 *Consider an augmented flippable execution of length $k \geq 0$ with t updaters, $\widehat{E}_k = U_0 s_1 S_1^* U_1 \ldots s_k S_k^* U_k$. If the read-only transactions by process q' are invisible, then the read-only transaction by process q does not terminate successfully.*

Proof Assume, towards a contradiction, that the read-only transaction of process q in $\widehat{E}_k$ terminates successfully and returns a value $(v_0, \ldots, v_{t-1})$, which does not violate serializability. Let the augmented flippable execution $\widehat{E}_k = U_0 s_1 S_1^* U_1 \ldots s_k S_k^* U_k$ correspond to a flippable execution $E_k = U_0 s_1 U_1 \ldots s_k U_k$.

By Lemma 7, the updates in $\widehat{E}_k$ are serialized in the order $U_0, U_1, \ldots, U_k$. The vector $(v_0, \ldots, v_{t-1})$ determines where q's read-only transaction is serialized. In particular, for some l, $0 < l \leq k$, the read-only transaction of q is serialized after U_{l-1} and before U_l, and for each item ι_f in $\{\iota_0 \ldots \iota_{t-1}\}$, either v_f is zero and no update wrote to ι_f before U_l, or the last update to ι_f before U_l wrote v_f to ι_f. Let S be the serialization of execution $\widehat{E}_k$.

Since the read-only transactions executed by process q' are invisible, $\widehat{E}_k$ and E_k are indistinguishable to $p_0, \ldots, p_{t-1}$ and q. Thus, they will execute the same steps in

both executions. Note that S is a serialization also for E_k. Since S preserves the real-time order among transactions, E_k is a flippable execution where the read-only transaction terminates and strict serializability is preserved, contradicting Lemma 1. □

As discussed before the lemma, the existence of a flippable execution (guaranteed by Lemma 3) implies there is an augmented flippable execution, and hence, Lemma 8 implies the following impossibility result:

Theorem 5 *There is no weakly disjoint-access parallel STM implementation with invisible read-only transactions of a serializable STM, in which read-only transactions always terminate successfully.*

When a read-only transaction of $t \geq 2$ data items applies non-trivial primitives to at most $t-2$ base objects, the read-only transactions of q' in the augmented flippable execution are, in fact, invisible since their read set contains only two data items. As discussed before Lemma 8, the existence of a flippable execution (guaranteed by Lemma 5) implies there is an augmented flippable execution, and hence, Lemma 8 implies the following lower bound:

Theorem 6 *In a serializable disjoint-access parallel STM implementation for $t+2$ processes, where all read-only transactions terminate successfully, some read-only transaction of $t > 2$ data items applies non-trivial primitives to at least $t-1$ base objects.*

The impossibility result and lower bound also hold for *virtual world consistency*, recently proposed by Imbs et al. [19]. This consistency condition requires serializability or strict serializability of committed transactions, and ensures that aborted transactions always see a consistent state of the memory, although not necessarily consistent with each other. Since our results do not consider the behavior of aborted transactions, they also hold for virtual world consistency.

5 Related Work

Many STM implementations are centralized; in particular, *Lazy Snapshot Algorithm* (LSA) [26] relies on a single shared monotonically increasing counter to determine a unique commit timestamp for transactions, while *Transactional Locking II* (TL2) [7] relies on a global clock. The former approach introduces a single hot-spot accessed by all transactions, regardless of their data sets, and is therefore not disjoint-access parallel, while the latter approach relies on a single source of synchronous time, which is not realistic for systems with a larger number of processes.

More recently, two STM implementations without a centralized hot-spot have been proposed. Avni and Shavit [4] present a *thread-local clock* mechanism that provides a decentralized solution for maintaining a consistent view. The key idea is using Lamport clock (scalar causal timestamps) instead of the real-time global clock. Integrated with TL2, this mechanism provides an STM supporting invisible read-only transactions, without a centralized contention point. A drawback of this algorithm is that

transactions that terminated long before the current one may cause it to fail since the timestamp recorded for them is not current enough. Thus, read-only transactions are not wait-free. Imbs and Raynal [17] propose an opaque lock-based STM with no centralized hot-spot but their solution has visible reads.

Guerraoui and Kapalka [9] prove that obstruction-free implementations of software transactional memory cannot ensure *strict* disjoint-access parallelism. This property requires transactions with disjoint data sets not to access a common base object. This notion is stronger than the one originally proposed by Israeli and Rappoport [20], and commonly used in the literature [14], where two transactions with disjoint data sets are allowed to access the same base objects, provided they are connected via other transactions. All other transactions have to progress in parallel, even if they are concurrent. Their definition of strict disjoint-access parallelism, like our first definition (Definition 1), allows concurrent reads to the same base objects even by transactions that are not connected in the conflict graph.

Our lower bound applies to the notion of disjoint-access parallelism as originally defined in [20]. In contrast, the result of Guerraoui and Kapalka [9] does not hold for this weaker requirement. Indeed, Herlihy et al. [16] present an obstruction-free and disjoint-access parallel STM. Obstruction-freedom does not prevent interfering concurrent processes from starving each other and thus, the implementation presented in [16] does not guarantee that a read-only transaction eventually terminates successfully.

Elsewhere, Guerraoui and Kapalka [10] prove a lower bound on the number of steps a process takes to successfully terminate a transaction, for every implementation that uses invisible reads, is single-version, and never aborts a transaction unless it conflicts with another live transaction. Our lower bound allows multi-version implementations, but requires read-only transactions to terminate successfully, regardless of overlapping transactions.

Serializability provides a weaker guarantee on the ordering of transactions (it does not have to respect the real-time order of non-overlapping ones). Nevertheless, our impossibility results hold also for serializable STMs that preserve the per-process order. Indeed, none of the serializable STM implementations presented in the literature, e.g., [5, 8, 24, 28], provides disjoint-access parallelism and has wait-free, invisible read-only transactions. In fact, the impossibility results hold also for STMs that satisfy the even weaker condition of *snapshot isolation* known from the database literature [23, 29] and suggested as an efficient alternative to serializability for STMs [27].

Riegel et al. [28] proposed an STM implementation that supports invisible reads and is disjoint-access parallel, but it provides only *causal serializability*; moreover, read-only transactions may abort infinitely many times. Causal serializability is weaker than serializability since it allows different processes to have a different view of the system. This leaves open the question of whether our results holds for causally serializable STMs, or whether the algorithm of [28] can be extended to have wait-free read-only transactions.

A read-only transaction can be considered as a *partial* scan operation [3]: a *partial snapshot object* is an atomic snapshot object [1], where processes can scan any subset of the components. In the wait-free algorithms for partial snapshot objects [3, 18],

scanners announce which components they are currently attempting to scan, i.e., read-only transactions are visible.

Our proof techniques draw ideas from the lower bounds on the step complexity of update operations in snapshot objects. Israeli and Shirazi [21] prove an $\Omega(m)$ lower bound on the number of steps to update a component in an m-component single-writer snapshot objects, implemented from single-writer registers. Attiya, Ellen and Fatourou [2] extend this lower bound to implementations of m-component multi-writer objects from base objects of any type.

6 Discussion

This paper shows that no transactional memory implementation can be disjoint-access parallel and have invisible, wait-free read-only transactions. There are implementations that are disjoint-access parallel and have invisible but not wait-free read-only transactions [4, 16], while others have invisible, wait-free read-only transactions but are not disjoint-access parallel [26]. In principle, the invisibility of read-only transactions can also be sacrificed in order to keep them wait-free, and the implementation disjoint-access parallel. This can be done by treating the read set together with the write set and adapting a dynamic disjoint-access parallel implementation of multi-location synchronization operator. For example, Harris et al. [13] give a disjoint-access parallel implementation of a multi-word compare-and-swap operation: an operation acquires locks on the words it needs, one by one; if another operation already holds the lock on a word, the operation helps the conflicting operation, thereby guaranteeing progress. (This algorithm is not wait-free, but additional helping can make it wait-free without sacrificing the other properties.) Thus, each of the assumptions made in our impossibility result is necessary, since removing either of them admits an implementation with the two remaining properties.

The multi-word compare-and-swap of [13], can be used to atomically validate the read set and update the write set of a transaction; it requires locking all items in the data set of the transaction. This application of a multi-word operation demonstrates our lower bound: A read-only transaction has to lock all items in its read set, and therefore it writes to distinct base objects representing these locks.

Our work joins recent efforts to explore the boundaries of STM implementations, so as to guide algorithm designers in their attempt to find better and more efficient implementations. Such boundaries demonstrate which directions are futile and which might lead to performance gains. It would be interesting to derive additional quantitative results on the complexity of transactions, and in particular, read-only transactions.

Our proof shows that the read-only transaction cannot terminate successfully, but it is possible to terminate it unsuccessfully, by *aborting* it; however, this abort is not justified by data conflicts. Moreover, when the read-only transaction is retried, it is possible to continue the construction and force it to abort again. An implementation is *permissive* with respect to a safety property [12] if it never aborts a transaction unless necessary for ensuring correctness. Our proof shows that a disjoint-access parallel implementation with invisible read-only transactions that always terminate—however,

not always successfully—is not permissive with respect to opacity, strict serializability, serializability or snapshot isolation. We would like to further investigate the connections between our results and the study of *unnecessary aborts* [8, 12, 22] or *wasted work* in STM implementations.

Acknowledgements We would like to thank Faith Ellen, Panagiota Fatourou, Rachid Guerraoui, Michal Kapalka, Martin Vechev and the referees for helpful comments.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. Assoc. Comput. Mach. **40**(4), 873–890 (1993)
2. Attiya, H., Ellen, F., Fatourou, P.: The complexity of updating multi-writer snapshot objects. In: Proc. 8th International Conference on Distributed Computing and Networking, pp. 319–330 (2006)
3. Attiya, H., Guerraoui, R., Ruppert, E.: Partial snapshot objects. In: Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 336–343 (2008)
4. Avni, H., Shavit, N.: Maintaining consistent transactional states without a global clock. In: Proc. 15th International Colloquium on Structural Information and Communication Complexity, pp. 131–140 (2008)
5. Aydonat, U., Abdelrahman, T.: Serializability of transactions in software transactional memory. In: 3rd ACM SIGPLAN Workshop on Transactional Computing (2008)
6. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. SIGMOD Rec. **24**(2), 1–10 (1995)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proc. 20th International Symposium on Distributed Computing, pp. 194–208 (2006)
8. Gramoli, V., Harmanci, D., Felber, P.: Towards a theory of input acceptance for transactional memories. In: Proc. 13th International Conference on Principle of Distributed Systems, pp. 527–533 (2008)
9. Guerraoui, R., Kapalka, M.: On obstruction-free transactions. In: Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 304–313 (2008)
10. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184 (2008)
11. Guerraoui, R., Kapalka, M.: The semantics of progress in lock-based transactional memory. In: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 404–415 (2009)
12. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: Proc. 22nd International Symposium on Distributed Computing, pp. 305–319 (2008)
13. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Proc. 16th International Symposium on Distributed Computing, pp. 265–279 (2002)
14. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Mateo (2008)
15. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
16. Herlihy, M., Luchangco, V., Moir, M.: Scherer III W. N.: Software transactional memory for dynamic-sized data structures. In: Proc. 22nd ACM Symposium on Principles of Distributed Computing, pp. 92–101 (2003)
17. Imbs, D., Raynal, M.: A lock-based protocol for software transactional memory. In: Proc. 13th International Conference on Principle of Distributed Systems, pp. 226–245 (2008)
18. Imbs, D., Raynal, M.: Help when needed, but no more: efficient read/write partial snapshot. In: Proc. 23nd International Symposium on Distributed Computing, pp. 142–156 (2009)
19. Imbs, D., Raynal, M., de Mendivil, J.R.: Brief announcement: virtual world consistency: a new condition for STM systems. In: Proc. 28th ACM Symposium on Principles of Distributed Computing, pp. 280–281 (2009)
20. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: Proc. 13th ACM Symposium on Principles of Distributed Computing, pp. 151–160 (1994)
21. Israeli, A., Shirazi, A.: The time complexity of updating snapshot memories. Inf. Process. Lett. **65**(1), 33–40 (1998)

22. Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. In: Proc. 21th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 59–68 (2009)
23. Lu, S., Bernstein, A., Lewis, P.: Correct execution of transactions at different isolation levels. IEEE Trans. Knowl. Data Eng. **16**(9), 1070–1081 (2004)
24. Napper, J., Alvisi, L.: Lock-free serializable transactions. Technical Report TR-05-04, The University of Texas at Austin (2005)
25. Papadimitriou, C.H.: The serializability of concurrent database updates. J. Assoc. Comput. Mach. **26**(4), 631–653 (1979)
26. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proc. 20th International Symposium on Distributed Computing, pp. 284–298 (2006)
27. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: 1st ACM SIGPLAN Workshop on Transactional Computing (2006)
28. Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: Proc. 26th ACM Symposium on Principles of Distributed Computing, pp. 340–341 (2007)
29. Weikum, G., Vossen, G.: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, San Mateo (2001)