Snapshot Isolation Does Not Scale Either

Victor Bushkov  Panagiota Fatourou
EPFL, IC, LPD  CARV, FORTH-ICS * & University of Crete

Dmytro Dziuma  Rachid Guerraoui
CARV, FORTH-ICS *  EPFL, IC, LPD


This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project.

*Computer Architecture & VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation of Research and Technology – Hellas (FORTH)
1 Introduction

Transactional memory (TM) [20, 25, 33] allows concurrent processes to execute operations on data items within atomic blocks of instructions, called transactions. The paradigm is appealing for its simplicity but implementing it efficiently is challenging. Ideally the TM system should not introduce any contention between transactions beyond that inherently due to the actual code of the transactions. In other words, if two transactions access disjoint sets of data items, then none of these transactions should delay the other one, i.e., these transactions should not contend on any base object. This requirement has been called strict disjoint-access-parallelism. Base objects are low-level objects, which provide atomic primitives like read/write, load linked/store conditional, compare-and-swap, used to implement the TM system. Two transactions contend on some base object if both access that object during their executions and one of them performs a non-trivial operation on that object, i.e., an operation which updates its state.

Disjoint-access-parallelism is met in the literature [2, 8, 15, 19, 26, 30] in many flavors (see the discussion in related work). Stronger versions of it, like strict disjoint-access-parallelism, result in more parallelism (and promote scalability) and therefore they are highly desirable when designing TM implementations: strict disjoint-access-parallelism is indeed ensured by blocking TM algorithms like TL [13]. Nevertheless, a transaction that locks a data item and gets paged out might block all other transactions for a long amount of time. One might require a liveness property that prevents such blocking. It was shown however in [18] that a TM cannot ensure strict disjoint-access-parallelism if it also needs to ensure serializability [29] and obstruction-freedom [16, 14]. Obstruction-freedom ensures that a transaction can be aborted only when step contention is encountered during the course of its execution. Obstruction-freedom is weaker than lock-freedom or wait-freedom. It allows for designing simpler TM algorithms and therefore it has been given special attention in TM computing [23].

In this paper, we study the following question: can we ensure strict disjoint-access-parallelism and obstruction freedom if we weaken safety? In other words, is consistency indeed a major factor against scalability? We focus on snapshot isolation [10], a safety property which requires that transactions should be executed as if every read operation reads from some snapshot of the memory that was taken when the transaction started. Snapshot isolation is an appealing property for TM computing since it provides the potential to increase throughput for workloads with long transactions [31]. If the set of transactions is restricted to those that do not read data items that have previously written, snapshot isolation is a weaker property than strict serializability.

We prove that the answer is still negative. Namely, it is impossible to implement a TM which is strict disjoint-access-parallel and satisfies obstruction-freedom and snapshot isolation. To make our impossibility result stronger, we consider, for its proof, a weak snapshot isolation property which requires only that each transaction reads from some consistent snapshot of the memory taken when it starts, thus ignoring the extra constraint (met in the literature [10, 31] for snapshot isolation) that from two concurrent transactions writing to the same data item, only one can commit. Moreover, the result still holds if the system provides primitives that atomically access any set of (up to) \( k \) base objects, where \( k \) is any integer.

The proof of our impossibility result is based on indistinguishability arguments. The same is true for the impossibility result in [18] which however holds for serializable TM algorithms and has a less intricate proof. Specifically, our proof employs several executions and a big number of transactions (up to \((k+2)(k+1)^2 + k + 3\) transactions), which access a big number of data items in total in contrast to the proof in [18] which presents a simple execution involving only three transactions, one of which accessing four data items and the other two accessing two data items each. The main difficulty comes from the fact that the read operations of a transaction do not have to be serialized at the same point as its write operations. So, it is much harder to construct an execution which violates snapshot isolation. We end up constructing two legal executions, chosen from a big set of executions, where a read-only transaction must return the same values for the data items that it reads. We then prove that one of these two executions violates snapshot isolation.
We finally show how to circumvent the impossibility result for read-only transactions: mainly we show how we can get a simplified version of DSTM [14], called SI-DSTM, which satisfies a strong version of snapshot isolation, obstruction-freedom, and the following weaker disjoint-access-parallelism requirement: two operations (executed by concurrent transactions) on different data items, one of which is a read operation, never contend on the same base object, and two write operations on different data items contend on the same base object only if there is a chain of transactions starting with the transaction that performs one of these write operations and ending with the transaction that performs the other, such that every two consecutive transactions in the chain conflict, i.e., they access a common data item. We call this property that is satisfied by write transactions weak disjoint-access-parallelism. DSTM satisfies weak disjoint-access-parallelism for all transactions, while SI-DSTM satisfies strict disjoint-access-parallelism for read-only transactions and weak disjoint-access-parallelism for write transactions. SI-DSTM is significantly simpler than the original DSTM and exhibits some performance benefits in comparison to DSTM. Specifically, no read operation can ever abort an update transaction (as is the case in DSTM) and thus SI-DSTM achieves better throughput. Also, read and write operations interfere less (in accessing base objects) than in the original DSTM, so incurring less contention.

Related Work. Disjoint-access-parallelism was introduced in [26]. Later variants [2,8,15] employed the concept of a conflict graph. A conflict graph is a graph whose vertices represent transactions (or operations) performed in an execution α and an edge exists between two nodes if the corresponding transactions (operations) access the same data item in α. In most of these definitions, disjoint-access-parallelism requires any two transactions to contend on a base object only if there is a path in the conflict graph of the minimal execution interval that contains both transactions such that every two consecutive transactions in the path conflict. Different variants are met in the literature with the names disjoint-access-parallelism or weak disjoint-access-parallelism (most of them use different properties to restrict access to a base object by two processes performing a transaction/operation).

In [2,4,6,26], additional constraints are placed on the length of the path in the conflict graph, resulting on what is known as d-local contention property, where d is the upper bound on the length of the path. In [26], where disjoint-access-parallelism originally appeared, an additional constraint on the step complexity of each operation is provided in the definition. Attiya et al. [8] proved that no (weak) disjoint-access-parallel TM implementation can support wait-free and invisible read-only transactions; a read-only transaction does not perform writes on data items and an invisible transaction does not perform non-trivial operations on base objects when reading data items. The variant of disjoint-access-parallelism considered in [8] ensures that processes executing two transactions concurrently contend on a base object only if there is a path between the two transactions in the conflict graph. Although our impossibility result is proved for a stronger disjoint-access-parallelism property, it considers a much weaker progress property, i.e., obstruction freedom, and holds even for TM algorithms where read-only transactions are visible. The proof of the impossibility result in [8] employs indistinguishability arguments based on flippable executions [5] which is significantly different from the indistinguishability arguments used here.

Recent work [11] has proved that, if the TM algorithm does not have access to the code of each transaction, a property similar to wait-freedom, called local progress, cannot be ensured by any TM algorithm. In [15], it is proved that wait-freedom cannot be achieved even if this restriction is abandoned (given that each time a transaction aborts it restarts its execution, as it is usually the case in TM computing) if even a weak version of disjoint-access-parallelism, called feeble disjoint-access-parallelism must be ensured. Thus, to achieve even weaker forms of disjoint-access-parallelism, one must consider weaker progress properties as we do in this paper.

Pelerman et al. [30] proved that no TM can be strictly serializable, (weak) disjoint-access-parallelism, and MV-permissive. The impossibility result holds under the assumptions that the TM does not have access to the code of transactions and the code for reading and writing data items terminates within a finite number of steps. For disjoint-access-parallelism, Pelerman et al. [30]
considers the same variant as in [8]. A TM implementation satisfies MV-permissiveness if a transaction aborts only if it is a write transaction that conflicts with another write transaction. This impossibility result can be beaten [7] if the stated assumptions do not hold. Our impossibility result holds if the TM ensures only snapshot isolation and even if it is MV-permissive; we do not make any assumptions to prove our result.

Several software transactional memory implementations [33, 13, 17, 23, 27, 34] are disjoint-access-parallel: TL [13] ensures strict disjoint-access-parallelism; the rest satisfy weaker forms of disjoint-access-parallelism [5] and among them OSTM [17] is lock-free. The TM in [33] is also lock-free but it has been designed for static transactions that access a pre-determined set of memory locations. Linearizable universal constructions [21, 22] which ensure weaker versions of disjoint-access-parallelism than that provided by SI-DSTM are presented in [11, 9, 15, 35]. Barnes [9] implementation is lock-free. The universal construction in [15] ensures wait-freedom when applied to objects that have a bound on the number of data items accessed by each operation they support, and lock-freedom in other cases. Disjoint-access-parallel wait-free universal constructions when each operation accesses a fixed number of predetermined memory locations are provided in [2, 35].

Snapshot isolation was originally introduced as a safety property in the database world [10, 28] to increase throughput for long read-only transactions. In the concept of TM, snapshot isolation has been studied in [3, 12, 31, 32]. An STM algorithm, called SI-STM, which ensures snapshot isolation is presented in [31]. SI-STM employs a global clock mechanism and therefore, it is not disjoint-access-parallel. In [12], static analysis techniques to detect, at compile time, consistency anomalies that may arise when the TM algorithm satisfies snapshot isolation or other weak safety properties are presented. Snapshot isolation on TM for message-passing systems has been studied in [3].

2 Preliminaries

We consider an asynchronous system with n processes which communicate by accessing shared base objects. A base object provides atomic primitives, to access or modify its state. The system may support various types of base objects like read/write(R/W) registers, load-link/store-conditional (LL/SC), compare-and-swap (CAS), fetch-and-add (F&A) etc. A primitive which can change the state of an object is called non-trivial; otherwise, it is called trivial.

Transactional memory (TM) employs transactions to execute pieces of sequential code in a concurrent environment. Each piece of code contains accesses to pieces of data, called data items, that may to be accessed by several processes when the code is executed concurrently; so TM should synchronize accesses to data items. To achieve this, a TM algorithm usually provides a shared representation for each data item by using base objects. A transaction may either commit, in which case all its updates become visible to other transactions, or abort, in which case its updates are discarded.

A TM algorithm provides implementations for the routines ReadDI and WriteDI which are called to read or write data items, respectively. TM algorithms that cope with dynamic data, should also provide an implementation for createDI which is called to create new data items. In addition, a TM algorithm provides implementations for the routines BeginTr, CommitTr, and AbortTr, which are called when a transaction starts its execution, and when it tries to commit or abort, respectively. Each time a transaction calls one of these routines we say that it invokes a transactional operation; when the execution of the routine completes, a response is returned. We denote the invocation of CommitTr by a transaction T as commitT; the response to commitT can be either C_T (commit) or A_T (abort). We denote by (i) \textit{x.write(v)} the invocation of WriteDI for data item x with value v; it returns ok if the write was successful or A_T if the transaction that invoked it has to abort, (ii) \textit{x.read()} the invocation of ReadDI for data item x; it returns a value for x if the operation was successful or A_T if the transaction that invoked it has to abort. Also we denote by beginT the invocation of BeginTr by T.

A configuration is a vector that, for each process and for each base object, contains a component
storing the state of this process or base object. In an initial configuration, processes and base objects are in initial states. A step of a process consists of a single primitive on some base object, the response to that primitive, and zero or more local operations that are performed after the access and which may cause the internal state of the process to change; each step is executed atomically. An execution $\alpha$ is a sequence of steps. An execution is legal starting from a configuration $C$ if the sequence of steps performed by each process follows the algorithm for that process (starting from its state in $C$) and, for each base object, the responses to the operations performed on the object are in accordance with its specification (and the state of the object at configuration $C$). An execution interval of execution $\alpha$ is a subsequence of consecutive steps from $\alpha$. We use $\alpha \cdot \beta$ to denote the execution $\alpha$ immediately followed by the execution $\beta$ and say that $\alpha$ is a prefix of $\alpha \cdot \beta$. An execution is solo if every step is performed by the same process. Two executions $\alpha_1$ and $\alpha_2$ starting from configurations $C_1$ and $C_2$, respectively, are indistinguishable to some process $p$, if the state of $p$ is the same in $C_1$ and $C_2$, and the sequence of steps performed by $p$ (and thus also the responses it receives) are the same during both executions.

Fix an execution $\alpha$ in which a transaction $T$ is executed. The execution interval of $T$ in $\alpha$ is the subsequence of consecutive steps of $\alpha$ starting with the first step executed by any of the operations invoked by $T$ and ending with the last such step. A TM algorithm is obstruction-free if a transaction $T$ can be aborted only when other processes take steps during the execution interval of $T$.

A history $H$ is a sequence of invocations of transactional operations and their responses. Given an execution $\alpha$, we denote by $H_{\alpha}$ the sequence of invocations and responses performed by the transactions executed in $\alpha$. We denote by $H|T$ the longest subsequence of $H$ consisting only of invocations and responses of a transaction $T$. Transaction $T$ is in history $H$ if $H|T$ is not empty. History $H$ is well-formed if for every transaction $T$ in $H$ the following holds: (i) $H|T$ is a sequence of alternating invocations and responses starting with BeginTr() followed by ok, (ii) each read invocation in $H|T$ is followed either by a value or by $A_T$, (iii) each write invocation in $H|T$ is followed either by an ok response or by $A_T$, (iv) each invocation of CommitTr() in $H|T$ is followed by $C_T$ or $A_T$, (v) each invocation of AbortTr() in $H|T$ is followed by $A_T$, (vi) no invocation follows by $T$ after $C_T$ or $A_T$ in $H|T$. Herein, we consider only well-formed histories.

We say that $T$ commits (aborts) in $H$ if $H|T$ ends with $C_T$ ($A_T$, respectively). If $T$ does not commit or abort in $H$, then $T$ is live in $H$. $H$ is complete if it does not contain any live transactions. If $H|T$ ends with commitT, then $T$ is commit-pending. Transaction $T_1$ precedes transaction $T_2$ in $H$, if $T_1$ is not live in $H$ and $A_{T_1}$ or $C_{T_1}$ precedes the first invocation of $T_2$ in $H$. If $T_1$ does not precede $T_2$ in $H$ and $T_2$ does not precede $T_1$ in $H$, then $T_1$ and $T_2$ are concurrent in $H$. A history $H$ is sequential if no two transactions are concurrent in $H$.

Transaction $T$ is legal in a sequential history $H$, if every read invocation $x.read()$, whose response is not $A_T$, returns a value $v$ such that: (i) if there exists an invocation of $x.write(*)$ by a committed transaction or by $T$ itself preceding $x.read()$, then $v$ is the argument of the last such $x.write(*)$ invocation; (ii) otherwise, $v$ is the initial value of $x$. A complete sequential history $H$ is legal if every transaction in $H$ is legal.

We say that two transactions conflict in an execution $\alpha$, if they both invoke a transactional operation on a common data item in $H_{\alpha}$. The conflict graph of an execution interval $I$ of $\alpha$ is an undirected graph whose vertices represent transactions that take steps in $I$ and an edge connects two transactions $T_1$ and $T_2$ iff $T_1$ conflicts with $T_2$ in $\alpha$. We say that two executions contend on a base object $o$ if they both contain a primitive on $o$ and one of these primitives is non-trivial.

Denote by $\alpha | T$ the subsequence of $\alpha$ consisting of all steps executed by $T$. A TM implementation $I$ is strict disjoint-access-parallel [13], if in each execution $\alpha$ of $I$, and for every two transactions $T_1$ and $T_2$ executed in $\alpha$, $\alpha | T_1$ and $\alpha | T_2$ contend on some base object, only if there is an edge between $T_1$ and $T_2$ in the conflict graph of $\alpha$. A TM implementation $I$ is weak disjoint-access-parallel, if in each execution $\alpha$ of $I$, and for every two transactions $T_1$ and $T_2$ executed in $\alpha$, $\alpha | T_1$ and $\alpha | T_2$ contend on some base object, only if there is a path between $T_1$ and $T_2$ in the conflict graph of the minimal execution interval of $\alpha$ containing $\alpha | T_1$ and $\alpha | T_2$.
Let $T$ be a committed or commit-pending transaction in a history $H$. A read operation $x.read()$ on some data item $x$ by $T$ is \emph{global} if $T$ has not invoked $x.write(*)$ before invoking $x.read()$. Let $T \mid read_g$ be the longest subsequence of $H \mid T$ consisting only of the global read invocations (and any matching responses) and $T \mid other$ be the subsequence $H \mid T \setminus T \mid read_g$, i.e. $T \mid other$ consists of all invocations performed by $T$ (and any matching responses) other than those comprising $T \mid read_g$. Let $\lambda$ be the empty execution. Then we define $T_g$ and $T_o$ in the following way:

- $T_g = \text{begin}_T \cdot \text{ok} \cdot T \mid \text{read}_g \cdot \text{commit}_{T_g} \cdot C_{T_g}$ if $T \mid \text{read}_g \neq \lambda$, and $T_g = \lambda$ otherwise, and
- $T_o = \text{begin}_T \cdot \text{ok} \cdot T \mid \text{other} \cdot \text{commit}_{T_o} \cdot C_{T_o}$ if $T \mid \text{other} \neq \lambda$, and $T_o = \lambda$ otherwise.

\textbf{Definition 2.1} (Snapshot Isolation). An execution $\alpha$ satisfies snapshot isolation, if for every committed transaction $T$ (and for some of the commit-pending transactions) in $\alpha$ it is possible to insert a read serialization point $*_{T,g}$ and a write serialization point $*_{T,o}$ such that: (i) $*_{T,g}$ precedes $*_{T,o}$, (ii) both $*_{T,g}$ and $*_{T,o}$ are inserted within the execution interval of $T$, and (iii) if $\sigma_\alpha$ is the sequence defined by these serialization points, in order, and $H_{\sigma_\alpha}$ is the history we get by replacing each $*_{T,g}$ with $T_g$ and each $*_{T,o}$ with $T_o$ in $\sigma_\alpha$, then $H_{\sigma_\alpha}$ is legal.

We note that this variant of snapshot isolation is strictly weaker than \emph{strict serializability} \cite{29} and thus also than \emph{opacity} \cite{19}. Roughly speaking, for every history $H$ that satisfies strict serializability and any committed transaction $T$ in $H$, both $*_{T,g}$ and $*_{T,o}$ can be inserted in the place of the serialization point for $T$.

Now, let $T \mid read$ be the longest subsequence of $H \mid T$ consisting only of read invocations and their corresponding responses and $T \mid write$ be the longest subsequence of $H \mid T$ consisting only of write invocations and their corresponding responses. Then we define $T_r$ and $T_w$ in the following way:

- $T_r = \text{begin}_T \cdot \text{ok} \cdot T \mid \text{read} \cdot \text{commit}_{T_r} \cdot C_{T_r}$ if $T \mid \text{read} \neq \lambda$, and $T_r = \lambda$ otherwise, and
- $T_w = \text{begin}_T \cdot \text{ok} \cdot T \mid \text{write} \cdot \text{commit}_{T_w} \cdot C_{T_w}$ if $T \mid \text{write} \neq \lambda$, and $T_w = \lambda$ otherwise.

\textbf{Definition 2.2} (R/W-independent Snapshot Isolation). An execution $\alpha$ satisfies R/W-independent snapshot isolation, if for every committed transaction $T$ (and for some of the commit-pending transactions) in $\alpha$ it is possible to insert a read serialization point $*_{T_r}$ and a write serialization point $*_{T,w}$ such that: (i) $*_{T,r}$ precedes $*_{T,w}$, (ii) both $*_{T,r}$ and $*_{T,w}$ are inserted within the execution interval of $T$, and (iii) if $\sigma_\alpha$ is the sequence defined by these serialization points, in order, and $H_{\sigma_\alpha}$ is the history we get by replacing each $*_{T,r}$ with $T_r$ and each $*_{T,w}$ with $T_w$ in $\sigma_\alpha$, then $H_{\sigma_\alpha}$ is legal.

R/W-independent snapshot isolation is incomparable to serializability \cite{29}, \emph{strict serializability} \cite{29}, and \emph{opacity} \cite{19}. For example, a solo execution of some transaction which updates a data item and then reads the new value of that data item satisfies serializability, but does not satisfy snapshot isolation. And a solo execution of some transaction which updates a data item and then reads the old value of that data item satisfies snapshot isolation, but does not satisfy serializability.

We remark that snapshot isolation and R/W-independent snapshot isolation, as defined above, do not satisfy prefix-closure. However, we can make them prefix-close if we require each prefix of $\alpha$ to satisfy the stated properties.

\section{Impossibility Result}

In this section we present our impossibility result i.e. we prove that it is impossible to construct a STM satisfying obstruction-freedom, strict disjoint-access-parallelism and any of the variants of snapshot isolation defined earlier: snapshot isolation or R/W-independent snapshot isolation. As it is shown later, the proof employs transactions that only read or only write. In this case, the definitions of snapshot isolation and R/W-independent snapshot isolation are merely identical.
3.1 Simple Case

For simplicity, we first provide the proof for the restricted case where a primitive can access at most 2 base objects. The proof for the general case when primitives can access \( k \geq 1 \) base objects is a natural generalization of this proof and is provided later.

**Theorem 3.1.** No obstruction-free STM can ensure both snapshot isolation and strict disjoint-access-parallelism. This holds even if the system provides primitives that can atomically access 2 base objects.

**Proof.** Suppose there is an obstruction-free STM which ensures both snapshot isolation and strict disjoint-access-parallelism. We start by describing the general strategy of the proof. For the proof we will employ some transactions, all executed by distinct processes: (1) for each \( 1 \leq i \leq 3 \), transaction \( T_i \) (executed by \( p_i^1 \)) writes value 1 to \( a_i^1, a_i^2, b_i^1, b_i^2, b_i^3, d_i \); (2) for each \( 1 \leq i \leq 7 \), transaction \( T_i \) (executed by \( p_i^2 \)) writes value 2 to \( a_i^1, a_i^2, a_i^3, c_i^1, c_i^2, c_i^3, e_i \); (3) for each \( 1 \leq i \leq 3 \) and \( 1 \leq j \leq 3 \), transaction \( T_{ij} \) reads from \( b_j \); (4) for each \( 1 \leq i \leq 7 \) and \( 1 \leq j \leq 3 \), transaction \( T_{ij} \) reads from \( c_j \).

We will construct two executions: \( \alpha = \alpha_2 \cdot s_i^1 \cdot \beta_3^1 \cdot \beta_4^1 \cdot \gamma_1 \cdot \gamma_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \gamma_5 \) and \( \beta = \alpha_2 \cdot \gamma_3 \cdot \gamma_4 \cdot \gamma_5 \), where \( x \) and \( y \), \( 1 \leq x \leq 7 \), \( 1 \leq y \leq 3 \), are indices to be determined later that indicate transactions \( T_x \) and \( T_x \) which have some desired properties among the sets of \( T_x \), \( 1 \leq i \leq 3 \), and \( T_x \), \( 1 \leq j \leq 7 \). Similarly, indices \( z_1 \) and \( z_2 \), \( 1 \leq z_1, z_2 \leq 3 \), are to be determined later. Roughly speaking, (1) \( \alpha_2 \) is an execution where each of the processes \( p_1^i, p_2^i, p_3^i, p_4^i \) executes solo a part of the transactions \( T_1, T_2, T_3, T_4 \), respectively. (2) \( \beta_3^1, \beta_4^1, \gamma_3, \gamma_4 \) are solo executions of transactions \( T_{3i}, T_{4i}, T_{3j}, T_{4j} \), respectively, (3) finally, 2, 3, and \( \gamma_5 \) are solo executions of a new transaction \( T_5 \) by distinct process \( p_5 \) which reads the data item \( a_2^2, d_y, c_x \). We will prove that (1) \( \alpha_2 \) and \( \alpha_5 \) are legal executions, (2) executions \( \beta_3 \) and \( \gamma_5 \) are indistinguishable to process \( p_5 \), and (3) snapshot isolation is violated in either \( \beta_3 \) or \( \gamma_5 \). The proof is structured in steps.

**Step 1. Definition of executions \( \alpha_1 \), configurations \( C_1 \) and steps \( s_i^1 \).** For \( 0 \leq i \leq 3 \) we will inductively define a sequence of executions \( \alpha_1 \), steps \( s_i^1 \), and configurations \( C_i \). Let \( C_0 = C_0 \) (i.e., \( C_0 \) is an initial configuration), \( \alpha_0 \) be an empty execution and \( s_0^1 \) be an empty step. Fix any \( 1 \leq i \leq 3 \) and assume that \( \forall j \), \( 0 \leq j < i \), \( \alpha_j \), \( C_j \) and \( s_j^1 \) have been defined. Let transaction \( T_i^1 \) be executed solo (by \( p_i^1 \)) from configuration \( C_{i-1} \). Since \( p_i^1 \) runs solo, obstruction-freedom implies that \( T_i^1 \) eventually commits. Let \( C_i^1 \) be the configuration resulting from the execution of the last step of \( T_i^1 \). If transaction \( T_{3i}^1 \) is executed solo from \( C_{i-1} \), then in the resulting execution, \( T_{3i}^1 \) reads the value 1 written by \( T_1^1 \) for \( b_j \), otherwise snapshot isolation is violated. If transaction \( T_{3i}^1 \) is executed solo from configuration \( C_{i-1} \), then in the resulting execution, \( T_{3i}^1 \) reads 0 for \( b_i^1 \), otherwise snapshot isolation is violated. Thus, there exists a step \( s_i^1 \) in the solo execution of \( T_i^1 \) from \( C_{i-1} \), resulting in a configuration \( C_i^1 \), such that (1) if \( T_{3i}^1 \) is executed solo from the configuration just before \( s_i^1 \), then in the resulting execution, \( T_{3i}^1 \) reads 0 for \( b_i^1 \); (2) and if \( T_{3i}^1 \) is executed solo from \( C_i^1 \), then in the resulting execution \( T_{3i}^1 \) reads the value written by \( T_1^1 \) for \( b_i^1 \). If there are more than one such steps, let \( s_i^1 \) be the first.) Denote by \( \beta_i^1 \) the execution where \( T_i^1 \) is executed solo from \( C_{i-1} \) until \( p_i^1 \) is poised to execute \( s_i \). Let \( \alpha_i^1 = \alpha_{i-1} \cdot \beta_i^1 \) and \( C_i^1 \) be the configuration that results from \( \alpha_i^1 \).

**Step 2. Definition of executions \( \alpha_2 \), configurations \( C_2 \) and steps \( s_i^2 \).** Fix any \( 1 \leq i \leq 7 \). Let transaction \( T_2^1 \) be executed solo (by \( p_i^2 \)) from configuration \( C_2 \). Since \( p_i^2 \) runs solo, obstruction-freedom implies that \( T_2^1 \) eventually commits. Let \( C_2^1 \) be the configuration resulting from the execution of the last step of \( T_2^1 \). If transaction \( T_{4i}^1 \) is executed solo from \( C_2^1 \), then in the resulting execution, \( T_{4i}^1 \) reads the value 2 written by \( T_2^1 \) for \( c_i^1 \). If transaction \( T_{4i}^1 \) is executed solo from configuration \( C_1 \), then in the resulting execution, \( T_{4i}^1 \) reads 0 for \( c_i^1 \). Thus, there exists a step \( s_i^2 \) in the solo execution of \( T_2^1 \) from \( C_1 \), resulting in a configuration \( C_2^1 \), such that (1) if \( T_{4i}^1 \) is executed solo from the configuration just before \( s_i^2 \), then in the resulting execution, \( T_{4i}^1 \) reads 0 for \( c_i^1 \); (2) and if \( T_{4i}^1 \) is executed solo from \( C_2^1 \), then in the resulting execution \( T_{4i}^1 \) reads the value 2 written by \( T_2^1 \) for
Figure 1: Execution $\alpha_1$

$T_1^1$ runs solo until it’s poised to execute $s_1^1$

$T_3^1$ runs solo until it’s poised to execute $s_3^1$

$T_2^1$ runs solo until it’s poised to execute $s_2^1$

$T_3^1$ runs solo until it’s poised to execute $s_3^1$

Figure 2: Execution $\alpha_2$

$T_1^1$ runs solo until it’s poised to execute $s_1^1$

$T_3^1$ runs solo until it’s poised to execute $s_3^1$

$T_2^1$ runs solo until it’s poised to execute $s_2^1$

$T_4^1$ runs solo until it’s poised to execute $s_4^1$

Figure 3: Execution $\alpha_3$

$T_3^1$ runs solo until it commits

$T_3^3$ runs solo until it commits

$T_4^3$ runs solo until it commits

$T_3^3$ runs solo until it commits

Figure 4: Execution $\alpha_3'$

$T_3^1$ runs solo until it commits

$T_4^3$ runs solo until it commits

$T_3^3$ runs solo until it commits

$T_3^3$ runs solo until it commits

Figure 5: Execution $\alpha_5$

$T_3^2$ runs solo until it commits

$T_4^2$ runs solo until it commits

$T_3^2$ runs solo until it commits

$T_5^2$ runs solo until it commits

Figure 6: Execution $\alpha_5'$
such that of sets among $T$.

**Step 3. Definition of $x$.** Let $\alpha_1 = \alpha_i^1 \cdot \beta_1^0 \cdot \ldots \cdot \beta_2^0 \cdot \delta_1^0 \cdot \ldots \cdot \delta_2^0$ (Figure 1). We argue that $\alpha_1$ is legal. This is so since each pair of transactions among $T_2^1, \ldots, T_2^3$ access disjoint sets of transactional variables, so by strict disjoint-access-parallelism, they do not contend on any base object.

We will define $x$ so that process $p_i^0$ doesn’t write in $\alpha_1$ (and therefore in $\beta_2^0 \cdot s_2^0$) to any base object that is accessed in any of the steps $s_1^0, s_2^0, s_1^1$ (Property 1).

For $1 \leq i \leq 7$, denote by $A_i^1$ the set of base objects modified by $T_2^i$ in $\alpha_1$. Notice that for $1 \leq l, m \leq 7, l \neq m$, it holds that $A_{l}^1 \cap A_{m}^1 = \emptyset$. Moreover, since we consider the case where $k = 2, s_1^1, 1 \leq i \leq 3$, accesses a set $O_{i}^1$ of at most 2 base objects.

Let $O_1 = \bigcup_{l=1}^{3} O_{l}^1$; it follows that $O_1$ contains at most $2 \cdot 3 = 6$ base objects. Thus there exists an index $x, 1 \leq x \leq 7$, such that $A_x^1 \cap O_1 = \emptyset$.

**Step 4. Definition of $y$.** Let $\alpha_2 = \alpha_x^2 \cdot s_1^1 \cdot s_1^2$. Since each pair of transactions among $T_2^1, T_2^2, T_2^3$ do not conflict, by strict disjoint-access-parallelism, no pair of steps among $s_1^1, s_1^2, s_1^3$ contend. This and the definition of $x$ imply that $\alpha_2$ is legal. Moreover, if $A_1^1 \leq i \leq 3$, is the set of base objects modified by $T_2^i$ in $\alpha_2$, then for each $1 \leq l, m \leq 3, l \neq m$, it holds that $A_l^1 \cap A_m^1 = \emptyset$. $s_1^2$ accesses a set $O_{x}^2$ of at most 2 base objects, it follows that there exists an index $x, 1 \leq y \leq 3$, so that $A_y^1 \cap O_{x}^2 = \emptyset$. Notice that we defined $y$ so that process $p_i^0$ doesn’t write in $\alpha_2$ (and therefore in $\beta_2^0 \cdot s_1^2$) to any base object that is accessed in step $s_2^0$ (Property 2).

**Step 5. Definition of $z_1$.** Starting from $C_2^1$, let process $p_y^0$ execute one step. Since by definition of $x$, no object modified in $\beta_2^0$ is accessed in $s_1^2$, it follows that this step (by $p_y^0$) is $s_1^1$. Then let a set of (distinct) processes $p_1^0, p_2^0, p_3^0, p_4^0, p_4^0, p_4^0, p_1^0$ run solo (in this order) to execute $T_3^1, T_3^2, T_3^3, T_4^x, T_4^x, T_4^x, T_4^x, T_4^x$ respectively, until they commit (this will occur because of obstruction-freedom). Let $\beta_1^1 \cdot \beta_2^3 \cdot \beta_3^2 \cdot \beta_4^1 \cdot \beta_2^1$ be these solo executions by $p_1^0, p_2^0, p_3^0, p_4^0, p_4^0$ respectively. Let $\alpha_3 = \alpha_x^2 \cdot s_1^1 \cdot \beta_1^1 \cdot \beta_2^2 \cdot \beta_3^2 \cdot \beta_2^1 \cdot \beta_1^1$ (Figure 3).

Let $A_{1}^2, A_{2}^2, A_{3}^2, A_{4}^2, A_{4}^2, A_{4}^2, A_{4}^2$ be the sets of base objects modified in $\beta_1^1 \cdot \beta_2^2 \cdot \beta_3^2 \cdot \beta_4^1 \cdot \beta_2^1$, respectively.

We will define $z_1$ so that $A_{x}^1 \cap O_{x}^2 = \emptyset$, $A_{y}^1 \cap O_{x}^2 = \emptyset$; moreover, it holds that $T_3^{y,z_1}$ reads 1 for $b_{1}^y$, and $T_4^{x,z_1}$ reads 0 for $c_{1}^x$ in $\alpha_3$ (Property 3).

Since each pair of transactions among $T_1^4, T_4^x, T_4^x$ do not conflict, the intersection of any pair of sets among $A_1^2, A_2^2, A_3^2, A_4^2, A_4^2, A_4^2, A_4^2$ is empty. Since $O_{x}^2$ contains at most two base objects, there are at most two such sets that have a non-empty intersection with $O_{x}^2$. Thus, there exists an index $z_1, 1 \leq z_1 \leq 3$ such that $A_{z}^1 \cap O_{x}^2 = \emptyset$. Since $T_3^{y,z_1}$ and $T_3^{y,z_1}$ do not conflict, it holds also that $A_{x}^1 \cap O_{x}^2 = \emptyset$.

We prove that $T_3^{y,z_1}$ reads 1 for $b_{1}^y$. Notice that $T_3^{y,z_1}$ and $T_3^{y,z_1}$ do not conflict, so $T_3^{y,z_1}$ and $T_3^{y,z_1}$ do not contend. Moreover, among the set of transactions $T_3^1, T_3^2, T_3^3$, it is only $T_3^{y,z_1}$ that $T_3^{y,z_1}$ conflicts with. Thus, $\alpha_3$ is indistinguishable from $s_1^1 \cdot s_1^2$ to $p_2^0$. By definition of $s_1^2$, $T_3^{y,z_1}$ reads 1 for $b_{1}^y$ in $\alpha_3$. Since $s_1^1 \cdot s_1^2$ is indistinguishable from $\alpha_3$ to $p_1^0$, $T_3^{y,z_1}$ reads 1 for $b_{1}^y$ in $\alpha_3$. Thus, the write serialization point of $T_3^{y,z_1}$ should come before the read serialization point of $T_3^{y,z_1}$ in $\alpha_3$ and so before the read serialization points of $T_3^{y,z_1}$ and $T_3^{y,z_1}$. Thus, $T_3^{y,z_1}$ reads 1 for $b_{1}^y$ in $\alpha_3$.

Similarly, by strict disjoint-access-parallelism, transaction $T_4^{x}$ doesn’t access any base object modified in $s_1^1$ or in $\beta_1^1 \cdot \beta_2^1 \cdot \beta_4^1 \cdot \beta_4^1 \cdot \beta_2^1$. Thus $\alpha_3$ is indistinguishable from $p_2^0$ to $T_4^{x}$. By definition of $s_2^0$, transaction $T_4^{x}$ reads 0 for $c_{1}^x$ in $\alpha_3$. Thus, $T_4^{x}$ reads 0 for $c_{1}^x$ in $\alpha_3$. Thus, the write serialization point of $T_4^{x}$ in $\alpha_3$ should come after the read serialization point of $T_4^{x}$ and so after the read serialization points of $T_4^{x}$ and $T_4^{x}$. It follows that $T_4^{x}$ reads 0 for $c_{1}^x$ in $\alpha_3$.

Let $\alpha_4 = \alpha_x^2 \cdot s_1^1 \cdot \beta_2^3 \cdot \beta_2^3 \cdot \beta_2^3$. Since transactions $T_3^{x} \cdot T_3^{x}, T_3^{y}, T_3^{x}, T_3^{y}$ do not conflict with each other, and the same is true for $T_4^{x}, T_4^{x}, T_4^{x}, T_4^{x}$, by strict disjoint-access-parallelism, it follows that $\alpha_4$ is legal. Executions $\alpha_3$ and $\alpha_4$ are indistinguishable for processes $p_3^3$ and $p_4^3$. So, it holds that transaction $T_3^{y,z_1}$ reads 1 for $b_{1}^y$ and $T_4^{x}$ reads 0 for $c_{1}^x$ in $\alpha_4$. 9
Step 6. Definition of $z_2$. Starting from $C_2$, let process $p_5^2$ execute $s_2^5$. Then let each of $p_1^3, p_2^3, p_3^3, p_5^3, p_1^2$ run solo (in this order) to execute $T_{x,2}, T_{x,3}, T_{y,2}, T_{y,3}$, respectively, until they commit (this will occur because of obstruction-freedom). Let $\gamma^1, \gamma^2, \gamma^3, \gamma^4$ be these solo executions by $p_1^3, p_2^3, p_3^3, p_5^3, p_1^2$, respectively. Let $\alpha_2' = \alpha_2' \cdot s_2' \cdot \gamma_2' \cdot \gamma_4' \cdot \gamma_2' \cdot \gamma_2' \cdot \gamma_2' \cdot \gamma_2'$ (Figure 4).

Let $B_3^1, B_3^2, B_3^3, B_3^4, B_3^5$ be the sets of base objects modified in $\gamma^1, \gamma^2, \gamma^3, \gamma^4, \gamma^5$ respectively. We will define $z_2$ so that $B_3^j \cap O_1^y = \emptyset, B_3^j \cap O_1^y = \emptyset$; moreover, it holds that $T_{x,2}^3$ reads 2 for $c_{x,2}^2$ and $T_{y,2}^3$ reads 0 for $y_{x,2}$ in $\alpha_3$ (Property 4).

Since each pair of transactions among $T_{x,1}^y, T_{x,2}^y, T_{y,3}^y$ do not conflict, the intersection of any pair of sets among $B_3^1, B_3^2, B_3^3$ is empty. Since $O_1^y$ contains at most 2 base objects, there are at most two such sets that have a non-empty intersection with $O_1^y$. Thus, there exists an index $z_2, 1 \leq z_2 \leq 3$ such that $B_3^z \cap O_1^y = \emptyset$. Since $T_{x,1}^y$ and $T_{x,2}^y$ do not conflict, it holds also that $T_{x,2}^y \cap O_1^y = \emptyset$.

We prove that $T_{x,2}^z$ reads 2 for $c_{x,2}^z$. Clearly, $\alpha_2' = \alpha_2' \cdot s_2' \cdot \gamma_2'$. By definition of $s_2^2$, $T_{x,1}^z$ reads 2 for $c_1^z$ in $\alpha_2' \cdot s_2'$. Since $\alpha_2' \cdot s_2'$ is indistinguishable from $\alpha_2' \cdot \gamma_2'$, $T_{x,2}^z$ reads 2 for $c_2^z$ in $\alpha_3$. Thus, the write serialization point of $T_{x,2}^z$ should come before the read serialization point of $T_{x,2}^z$ in $\alpha_3$ and so before the read serialization points of $T_{x,2}^z$ and $T_{x,2}^z$. Thus, $T_{x,2}^z$ reads 2 for $c_{x,2}^z$ in $\alpha_3$.

Similarly, by strict disjoint-access-parallelism, transaction $T_{y,3}^y$ doesn’t access any base object modified in $\beta_1^y, \ldots, \beta_3^y$ or in $\gamma_1^y, \gamma_2^y, \gamma_3^y, \gamma_2^y$. Thus $\alpha_3$ is indistinguishable from $\alpha_3^y$ to $p_5^3$. By definition of $s_1^3$, transaction $T_{y,1}^y$ reads 0 for $b_3^y$ in $\alpha_3^y$. Thus, $T_{y,2}^y$ reads 0 for $b_3^y$ in $\alpha_3^y$. Therefore, the write serialization point of $T_{y,1}^y$ in $\alpha_3^y$ should come after the read serialization point of $T_{y,1}^y$ and so after the read serialization points of $T_{x,3}^y$ and $T_{y,3}^y$. It follows that $T_{y,2}^y$ reads 0 for $b_3^y$ in $\alpha_3^y$.

Let $\alpha_4' = \alpha_2' \cdot s_1^y \cdot \gamma_2^y \cdot \gamma_2^y$. Since transactions $T_{x,1}^y, T_{y,2}^y, T_{y,3}^y, T_{x,2}^y, T_{y,3}^y$ do not conflict with each other, by strict disjoint-access-parallelism, it follows that $\alpha_4'$ is legal. Execution $\alpha_3^y$ is indistinguishable from $\alpha_4'$ to $p_3^2$ and $p_5^2$. So, it holds that transaction $T_{y,2}^z$ reads 2 for $c_{x,2}^z$ and $T_{y,2}^z$ reads 0 for $b_3^y$ in $\alpha_3^y$.

Step 7. Executions $\beta_5$ and $\gamma_5$ are indistinguishable to process $p_5$. Consider now the executions $\alpha_5 = \alpha_4 \cdot \gamma_5 = \alpha_5 \cdot \alpha_5 \cdot \beta_5 = \alpha_5 \cdot \beta_5' \cdot \beta_5' \cdot \beta_5 \cdot \alpha_5' = \alpha_4 \cdot \alpha_5 \cdot \gamma_5 = \alpha_5 \cdot \beta_5 \cdot \gamma_5$, and $\gamma_5 = \gamma_5$ (Figures 5 and 6). Recall that $\beta_5$ and $\gamma_5$ are solo executions of transaction $T_5$ (which is executed by process $p_5$ and reads the data items $a_5^y, d_5^y, e_5$) until $T_5$ commits (obstruction-freedom guarantees that this will occur). Recall also that $\alpha_5 = \beta_5' \cdot \beta_5' \cdot \beta_5' \cdot \beta_5'$. Since $T_5$ does not conflict with $T_{y,1}^y$ and $T_{x,2}^x (T_{x,2}^y$ and $T_{x,2}^z$), in executions $\beta_5' \cdot \beta_5' \cdot \beta_5' \cdot \beta_5'$, processes $p_3^x$ and $p_5^2 (p_3^z$ and $p_5^2$) do not modify any base object read in $\beta_5$ ($\gamma_5$). Moreover, by definition, steps $s_1^y$ and $s_2$ do not contend. It follows that $\beta_5$ is indistinguishable from $\gamma_5$ to $p_5$. Thus $T_5$ reads the same values for $a_5^y, d_5^y, e_5$ as in both $\alpha_5$ and $\alpha_5'$.

Step 8. Snapshot isolation is violated in either $\alpha_5$ or in $\alpha_5'$. Recall that $T_{y,2}^3$ reads 1 for $b_3^y$ in $\alpha_4$ so it reads 1 for $b_3^y$ in $\alpha_5$. Therefore, in $\alpha_5$, the write serialization point of $T_{y,2}^3$ precedes the read serialization point of $T_{x,2}^2$, and thus also that of $T_5$. So, $T_5$ reads 1 for $d_5$ in $\beta_5$ (notice that no transaction other than $T_{y,2}^3$ writes to $d_5$). Similarly, recall that $T_{y,2}^2$ reads 2 for $c_{x,2}^2$ in $\alpha_5$ so it reads 2 for $c_{x,2}^2$ in $\alpha_5'$. Therefore, in $\alpha_5'$, the write serialization point of $T_{x,2}^3$ precedes the read serialization point of $T_{x,2}^3$, and thus also that of $T_5$. So, $T_5$ reads 2 for $e_5$ in $\gamma_5$ (notice that no transaction other than $T_{x,2}^3$ writes to $e_5$). Since $\beta_5$ is indistinguishable from $\gamma_5$ to $p_5$, $T_5$ reads 1 for $d_5$ and $e_5$ for $x_5$ in both executions. Thus, the write serialization points of $T_{y,2}^3$ and $T_{x,2}^3$ are placed before the read serialization point of $T_5$ in both executions. Depending on which one of them is going first, $T_5$ reads either 1 or 2 for $a_5^y$. Assume that $T_5$ reads 1 for $a_5^y$ (the case that $T_5$ reads 2 for $a_5^y$ is "symmetric").

We argue that snapshot isolation is violated in $\alpha_5$. Recall that $T_{y,2}^3$ reads 1 for $b_3^y$ in $\alpha_5$, thus the write serialization point of $T_{y,2}^3$ must come before the read serialization point of $T_{y,2}^3$. $T_{y,2}^3$ finishes its execution before the beginning of $T_{x,2}^3$, so the read serialization point of $T_{y,2}^3$ precedes the read serialization point of $T_{x,2}^3$. Recall that $T_{y,2}^3$ reads 0 for $c_{x,2}^2$ in $\alpha_5$, and therefore also in $\alpha_3$. Thus, the read serialization point of $T_{x,2}^3$ must come before the write serialization point of $T_{x,2}^3$. It follows
that the write serialization point of $T_3^r$ precedes the write serialization point of $T_2^r$, which implies that transaction $T_3$ must read 2 for $a_3^y$. A contradiction.

3.2 General case

Theorem 3.2. No obstruction-free STM can ensure both snapshot isolation and strict disjoint-access-parallelism. This holds even if the system provides primitives that can atomically access $k$ base objects, where $k \geq 1$ is any integer.

Proof. Suppose there is an obstruction-free STM which ensures both snapshot isolation and strict disjoint-access-parallelism. We first describe the general strategy of the proof in the same way as it was done in the previous section. For the proof we will the following transactions, all executed by distinct processes: (1) for each $1 \leq i \leq k + 1$, transaction $T_i^1$ (executed by $p_i^1$) writes value 1 to $a_1^i, \ldots, a_{k(k+1)+1}^i, b_1^i, \ldots, b_{k+1}^i, d_i$; (2) for each $1 \leq i \leq k(k+1)+1$, transaction $T_2^i$ (executed by $p_2^i$) writes value 2 to $a_1^i, \ldots, a_{i(k+1)+1}^i, c_1^i, \ldots, c_{k+1}^i, e_i$; (3) for each $1 \leq i \leq k + 1$ and $1 \leq j \leq k + 1$, transaction $T_3^{ij}$ reads from $b_j^i$; (4) for each $1 \leq i \leq k(k+1)+1$ and $1 \leq j \leq k + 1$, transaction $T_4^{ij}$ reads from $c_j^i$.

We will construct two executions: $\alpha_5 = \alpha_5^x \cdot s_y^1 \cdot \beta_3^1 \cdot s_y^2 \cdot \beta_y$ and $\alpha_5' = \alpha_5^x \cdot s_y^2 \cdot \gamma_2^2 \cdot \gamma_4^2 \cdot s_y^1 \cdot \gamma_5$, where $x$ and $y, 1 \leq x \leq k(k+1)+1$, $1 \leq y \leq k + 1$, are indices to be determined later which indicate transactions $T_1^i$ and $T_2^i$ which have some desired properties among the sets of $T_1^i, 1 \leq i \leq k + 1$, and $T_2^i, 1 \leq j \leq k(k+1)+1$. Similarly, indices $z_1$ and $z_2, 1 \leq z_1, z_2 \leq k + 1$, are to be determined later. Roughly speaking, (1) $\alpha_5^x$ is an execution where each of the processes $p_1^1, \ldots, p_i^k+1, p_2^1$ executes solo a part of the transactions $T_1^i, \ldots, T_2^{i+1}, T_2^j$, respectively, (2) $\beta_3^1, \beta_4^1, \gamma_2^2, \gamma_4^2$ are solo executions of transactions $T_3^i, \gamma_3^i, T_3^i, \gamma_3^i, T_3^i, \gamma_3^i, T_3^i, \gamma_3^i$, and (3) $\beta_5$ and $\gamma_5$ are solo executions of a new transaction $T_5$ by distinct process $p_5$ which reads the data items $a_y^1, d_y, e_x$. We will prove that (1) $\alpha_5$ and $\alpha_5'$ are legal executions, (2) executions $\beta_5$ and $\gamma_5$ are indistinguishable to process $p_5$, and (3) snapshot isolation is violated in either $\beta_5$ or $\gamma_5$. The proof is structured in steps.

Step 1. Definition of executions $\alpha_1^i$, configurations $C_1^i$ and steps $s_1^i$. For $1 \leq i \leq k + 1$ we will inductively define a sequence of executions $\alpha_1^i$, steps $s_1^i$, and configurations $C_1^i$. Let $C_1^0 = C_0$ (i.e., $C_1^0$ is an initial configuration), $\alpha_1^0$ be an empty execution and $s_1^0$ be an empty step. Fix any $1 \leq i \leq k + 1$ and assume that $\forall j, 0 \leq j < i, \alpha_1^j, C_1^j$ and $s_1^j$ have been defined. Let transaction $T_1^i$ be executed solo (by $p_i^1$) from configuration $C_1^{i-1}$. Since $p_i^1$ runs solo, obstruction-freedom implies that $T_1^i$ eventually commits. Let $C_1^i$ be the configuration resulting from the execution of the last step of $T_1^i$. If transaction $T_3^{i-1}$ is executed solo from $C_1^i$, then in the resulting execution, $T_3^{i-1}$ reads the value 1 written by $T_1^i$ for $b_1^i$, otherwise snapshot isolation is violated. If transaction $T_3^{i-1}$ is executed solo from configuration $C_1^{i-1}$, then in the resulting execution, $T_3^{i-1}$ reads 0 for $b_1^i$, otherwise snapshot isolation is violated. Thus, there exists a step $s_1^i$ in the solo execution of $T_1^i$ from $C_1^{i-1}$, resulting in a configuration $C_1'^i$, such that (1) if $T_3^{i-1}$ is executed solo from the configuration just before $s_1^i$, then in the resulting execution, $T_3^{i-1}$ reads 0 for $b_1^i$; (2) and if $T_3^{i-1}$ is executed solo from $C_1'^i$, then in the resulting execution $T_3^{i-1}$ reads the value 1 written by $T_1^i$ for $b_1^i$. (If there are more than one such steps, let $s_1^i$ be the first.) Denote by $\beta_1^i$ the execution where $T_1^i$ is executed solo from $C_1^{i-1}$ until $p_i^1$ is poised to execute $s_1^i$. Let $\alpha_1^i = \alpha_1^{i-1} \cdot \beta_1^i$ and let $C_1^i$ be the configuration that results from $\alpha_1^i$.

Step 2. Definition of executions $\alpha_2^i$, configurations $C_2^i$ and steps $s_2^i$. Fix any $1 \leq i \leq k(k+1)+1$. Let transaction $T_2^i$ be executed solo (by $p_2^i$) from configuration $C_1^{k+1}$. Since $p_2^i$ runs solo, obstruction-freedom implies that $T_2^i$ eventually commits. Let $C_2^i$ be the configuration resulting from the execution of the last step of $T_2^i$. If transaction $T_4^{i-1}$ is executed solo from $C_2^i$, then in the resulting execution, $T_4^{i-1}$ reads the value 2 written by $T_2^i$ for $c_1^i$. If transaction $T_4^{i-1}$ is executed solo from configuration $C_1^{k+1}$, then in the resulting execution, $T_4^{i-1}$ reads 0 for $c_1^i$. Thus, there exists a step $s_2^i$ in the solo execution of $T_2^i$ from $C_1^{k+1}$, resulting in a configuration $C_2'^i$, such that (1) if $T_4^{i-1}$ is executed solo
\[ \beta_{1}^{1} \quad T_{1}^{1} \text{ runs solo until it's poised to execute } s_{1}^{1} \]
\[ \beta_{1}^{k+1} \quad T_{1}^{k+1} \text{ runs solo until it's poised to execute } s_{1}^{k+1} \]
\[ \beta_{2}^{1} \quad T_{2}^{1} \text{ runs solo until it's poised to execute } s_{2}^{1} \]
\[ \beta_{2}^{k+1} \quad T_{2}^{k+1} \text{ runs solo until it's poised to execute } s_{2}^{k+1} \]

Figure 7: Execution $\alpha_1$

\[ \beta_{1}^{1} \quad T_{1}^{1} \text{ runs solo until it's poised to execute } s_{1}^{1} \]
\[ \beta_{1}^{k+1} \quad T_{1}^{k+1} \text{ runs solo until it's poised to execute } s_{1}^{k+1} \]
\[ \beta_{2}^{1} \quad T_{2}^{1} \text{ runs solo until it's poised to execute } s_{2}^{1} \]
\[ \beta_{2}^{k+1} \quad T_{2}^{k+1} \text{ runs solo until it's poised to execute } s_{2}^{k+1} \]

Figure 8: Execution $\alpha_2$

\[ \alpha_{2}^{x} \quad s_{1}^{y} \quad \beta_{3}^{1} \quad T_{3}^{y,1} \text{ runs solo until it commits} \]
\[ \beta_{3}^{k+1} \quad T_{3}^{y,k+1} \text{ runs solo until it commits} \]
\[ \beta_{4}^{k+1} \quad T_{4}^{x,k+1} \text{ runs solo until it commits} \]
\[ \beta_{4}^{1} \quad T_{4}^{x,1} \text{ runs solo until it commits} \]

Figure 9: Execution $\alpha_3$

\[ \alpha_{2}^{x} \quad s_{2}^{x} \quad \gamma_{4}^{1} \quad T_{4}^{x,1} \text{ runs solo until it commits} \]
\[ \gamma_{4}^{k+1} \quad T_{4}^{x,k+1} \text{ runs solo until it commits} \]
\[ \gamma_{3}^{k+1} \quad T_{3}^{y,k+1} \text{ runs solo until it commits} \]
\[ \gamma_{3}^{1} \quad T_{3}^{y,1} \text{ runs solo until it commits} \]

Figure 10: Execution $\alpha'_3$

\[ \alpha_{2}^{y} \quad s_{1}^{y} \quad \beta_{3}^{z_1} \quad T_{3}^{y,z_1} \text{ runs solo until it commits} \]
\[ \beta_{3}^{z_1} \quad T_{4}^{x,z_1} \text{ runs solo until it commits} \]
\[ \beta_{5} \quad T_{5} \text{ runs solo until it commits} \]

Figure 11: Execution $\alpha_5$

\[ \alpha_{2}^{2} \quad s_{2}^{z_2} \quad \gamma_{3}^{2} \quad T_{3}^{y,z_2} \text{ runs solo until it commits} \]
\[ \gamma_{3}^{2} \quad T_{4}^{x,z_2} \text{ runs solo until it commits} \]
\[ \gamma_{5} \quad T_{5} \text{ runs solo until it commits} \]

Figure 12: Execution $\alpha'_5$
from the configuration just before $s_2$, then in the resulting execution, $T_{4}^{x_1}$ reads $0$ for $c_1$; (2) and if $T_{4}^{x_1}$ is executed solo from $C_{4}^{s_1}$, then in the resulting execution $T_{4}^{x_1}$ reads the value written by $T_{2}^{x_2}$ for $c_1$. (If there are more than one such steps, let $s_2$ be the first.) Denote by $\beta_2$ the execution where $T_{2}^{x}$ is executed solo from $C_{1}^{x_1}$ until $p_2$ is poised to execute $s_2$. Let $\alpha_2 = \alpha_1 \cdot \beta_2$ and let $C_2'$ be the configuration that results from $\alpha_2$.

**Step 3. Definition of $x$.** Let $\alpha_1 = \alpha_1^{k_1+1} \cdot \beta_2^{(k_1+1)\cdot s_1^{k_1}}, \ldots, s_1^{k_1}$ (Figure 4). We argue that $\alpha_1$ is legal. This is so since each pair of transactions among $T_1, \ldots, T_{4}^{(k_1+1)\cdot s_1^{k_1}}$ access disjoint sets of transactional variables, so by strict disjoint-access-parallelism, they do not contend on any base object.

We will define $x$ so that process $p_2^{x}$ doesn’t write in $\alpha_1$ (and therefore in $\beta_2^{(k_1\cdot s_2^{k_1})}$) to any base object that is accessed in any of the steps $s_1^{k_1}, \ldots, s_1^{k_1+1}$ (Property 1).

For $1 \leq i \leq k+1$, denote by $A_i$ the set of base objects modified by $T_{3}^{y}$ in $\alpha_1$. Notice that for $1 \leq i, m \leq (k+1) + 1, l \neq m$, it holds that $A_i \cap A_m = \emptyset$. Moreover, $s_1^{k_1}, 1 \leq i \leq k+1$, accesses a set $O_i$ of at most $k$ base objects.

Let $O_i = \bigcup_{l=1}^{k_1} O_{l}^{i}$; it follows that $O_i$ contains at most $k+1$ base objects. Thus there exists an index $x$, $1 \leq x \leq k+1 + 1$, such that $A_x \cap O_i = \emptyset$.

**Step 4. Definition of $y$.** Let $\alpha_2 = \alpha_2^{k_1+1} \cdot s_1^{k_1+1}$ (Figure 5). Recall that $\alpha_2 = \alpha_1^{k_1+1} \cdot \beta_2^{(k_1+1)\cdot s_1^{k_1}}$. Since each pair of transactions among $T_1, \ldots, T_{4}^{(k_1+1)\cdot s_1^{k_1}}$ do not conflict, by strict disjoint-access-parallelism, no pair of steps among $s_1^{k_1}, s_1^{k_1+1}$ contend. This and the definition of $x$ imply that $\alpha_2$ is legal. Moreover, if $A_1$, $1 \leq i \leq k+1$, is the set of base objects modified by $T_{3}^{y}$ in $\alpha_2$, then for each $1 \leq i, j \leq k+1, i \neq j$, it holds that $A_i \cap A_j = \emptyset$. Since $x^{\emptyset}$ accesses a set $O_x$ of at most $k$ base objects, it follows that there exists an index $y$, $1 \leq y \leq k+1$, so that $A_y^{\emptyset} \cap O_x = \emptyset$. Notice that we defined $y$ so that process $p_2^{y}$ doesn’t write in $\alpha_2$ (and therefore in $\beta_2^{y} \cdot s_2^{y}$) to any base object that is accessed in step $s_x^{y}$ (Property 2).

**Step 5. Definition of $z_1$.** Starting from $C_x^{y}$, let process $p_2^{y}$ execute one step. Since by definition of $x$, no object modified in $\beta_2^{y}$ is accessed in $s_x^{y}$, it follows that this step (by $p_2^{y}$) is $s_x^{y}$. Then let a set of (distinct) processes $p_3^{y}, \ldots, p_4^{y+k_1+1}, p_4^{y+k_1+1}$ run solo (in this order) to execute $T_3^{y}, \ldots, T_{4}^{y+k_1+1}, T_4^{x_1}, \ldots, T_{4}^{x_1}$, respectively, until they commit (this will occur because of obstruction-freedom).

Let $\beta_1^{y}, \ldots, \beta_3^{y}, \ldots, \beta_3^{k_1+1}, \beta_4^{y}, \ldots, \beta_4^{y+k_1+1}$ be these solo executions by $p_3^{y}, \ldots, p_3^{y+k_1+1}, p_4^{y+k_1+1}, \ldots, p_4^{y}$, respectively. Let $\alpha_3 = \alpha_2^{y} \cdot s_x^{y} \cdot \beta_1^{y} \cdot \beta_1^{k_1+1} \cdot \beta_4^{y} \cdot \beta_4^{y+k_1+1}$ (Figure 6).

Let $A_3^{y_1}, \ldots, A_3^{y+k_1+1}, A_4^{y_1}, \ldots, A_4^{y+k_1+1}$ be the sets of base objects modified in $\beta_3^{y}, \ldots, \beta_3^{k_1+1}, \beta_4^{y}, \ldots, \beta_4^{y+k_1+1}$, respectively. We will define $z_1$ so that $A_3^{z_1} \cap O_x = \emptyset, A_3^{z_1} \cap O_x = \emptyset$; moreover, it holds that $T_3^{y_1}$ reads $1$ for $b_3^{y}$ and $T_4^{x_1}$ reads $0$ for $c_3^{z_1}$ in $\alpha_3$ (Property 3).

Since each pair of transactions among $T_3^{y_1}, \ldots, T_{4}^{x_1}$ do not conflict, the intersection of any pair of sets among $A_1^{y_1}, \ldots, A_1^{y+k_1+1}$ is empty. Since $O_x$ contains at most $k$ base objects, there are at most $k$ such sets that have a non-empty intersection with $O_x$. Thus, there exists an index $z_1, 1 \leq z_1 \leq k+1$ such that $A_3^{z_1} \cap O_x = \emptyset$. Since $T_3^{y_1}$ and $T_3^{x_1}$ do not conflict, it holds also that $A_3^{z_1} \cap O_x = \emptyset$.

We prove that $T_3^{y_1}$ reads $1$ for $b_3^{y}$. Notice that $T_3^{y_1}$ and $T_3^{x_1}$ do not conflict, so $T_3^{y_1}$ and $T_3^{x_1}$ do not contend. Moreover, among the set of transactions $T_1^{y_1}, \ldots, T_{4}^{y_k+1}$, it is only $T_3^{y_1}$ that $T_3^{y_1}$ conflicts with. Thus, $\alpha_3$ is indistinguishable from $\alpha_3^{y} \cdot s_x^{y}$ to $p_3$. By definition of $s_x^{y}$, $T_3^{y_1}$ reads $1$ for $b_3^{y}$ in $\alpha_3^{y} \cdot s_x^{y}$. Since $\alpha_3^{y} \cdot s_x^{y}$ is indistinguishable from $\alpha_3$ to $p_3$, $T_3^{y_1}$ reads $1$ for $b_3^{y}$ in $\alpha_3$. Thus, the write serialization point of $T_3^{y_1}$ should come before the read serialization point of $T_3^{y_1}$ in $\alpha_3$ and so before the read serialization points of $T_3^{y_2}, T_3^{y_3}, \ldots, T_3^{y_k+1}$. Thus, $T_3^{y_1}$ reads $1$ for $b_3^{y}$ in $\alpha_3$.

Similarly, by strict disjoint-access-parallelism, transaction $T_{4}^{x_1}$ doesn’t access any base object modified in $s_2^{y}$ or in $\beta_3^{y}, \ldots, \beta_3^{k_1+1}, \beta_4^{y}, \ldots, \beta_4^{y+k_1+1}$. Thus $\alpha_3$ is indistinguishable from $\alpha_3^{y}$ to $p_4^{y}$. By definition of $s_2^{y}$, transaction $T_{4}^{x_1}$ reads $0$ for $c_1^{y}$ in $\alpha_3^{y}$. Therefore, $T_{4}^{x_1}$ reads $0$ for $c_1^{y}$ in $\alpha_3$. Thus,
the write serialization point of $T_2^x$ in $\alpha_3$ should come after the read serialization point of $T_4^x$ and so after the read serialization points of $T_4^{x,2}, \ldots, T_4^{x,k+1}$. It follows that $T_4^{y,z}$ reads 0 for $c_{22}^x$ in $\alpha_3$.

Let $\alpha_4 = \alpha_2^x \cdot s_1^y \cdot \beta_3^z \cdot s_2^z$. Since transactions $T_3^{y,1}, \ldots, T_3^{y,k+1}$ do not conflict with each other, and the same is true for $T_4^{x,1}, \ldots, T_4^{x,k+1}$, by strict disjoint-access-parallelism, it follows that $\alpha_4$ is legal. Executions $\alpha_3$ and $\alpha_4$ are indistinguishable for process $p_3^y$ and for process $p_1^y$. So, it holds that transaction $T_3^{y,z}$ reads 1 for $b_{32}^y$ and $T_4^{y,z}$ reads 0 for $c_{22}^y$ in $\alpha_4$.

**Step 6. Definition of $z_2$.** Starting from $C_2^y$, let process $p_2^y$ execute $s_2^y$. Then let each of $p_1^y, \ldots, p_5^y, \ldots, p_3^y$ run solo (in this order) to execute $T_4^{x,1}, \ldots, T_4^{x,k+1}, T_3^{y,k+1}, \ldots, T_3^{y,1}$, respectively, until they commit (this will occur because of obstruction-freedom). Let $\gamma_1^y, \ldots, \gamma_k^y, \gamma_k^{y+1}, \ldots, \gamma_3^{y+1}$ be these solo executions by $p_1^y, \ldots, p_5^y, \ldots, p_3^y$, respectively. Let $\alpha_3 = \alpha_2^x \cdot s_2^z \cdot \gamma_1^y \cdot \ldots \cdot \gamma_k^{y+1} \cdot \gamma_3^{y+1}$ (Figure 10).

Let $B_3^y, B_4^y, B_5^y$ be the sets of base objects modified in $\gamma_1^y, \ldots, \gamma_k^{y+1}, \gamma_3^{y+1}$, respectively. We will define $z_2$ so that $B_3^y \cap O_1^y = \emptyset, B_4^y \cap O_1^y = \emptyset$; moreover, it holds that $T_4^{y,z}$ reads 2 for $c_{22}^z$ and $T_3^{y,z}$ reads 0 for $b_{22}^z$ in $\alpha_3$ (Property 4).

Since each pair of transactions among $T_3^{y,1}, \ldots, T_3^{y,k+1}$ do not conflict, the intersection of any pair of sets among $B_3^y, \ldots, B_5^y$ is empty. Since $O_1^y$ contains at most $k$ base objects, there are at most $k$ such sets that have a non-empty intersection with $O_1^y$. Thus, there exists an index $x_2, 1 \leq x_2 \leq k+1$ such that $B_2^y \cap O_1^y = \emptyset$. Since $T_1^y$ and $T_4^{y,z}$ do not conflict, it holds also that $B_4^{y,x_2} \cap O_1^y = \emptyset$.

We prove that $T_4^{y,z}$ reads 2 for $c_{22}^z$. Clearly, $\alpha_3$ is indistinguishable from $\alpha_2^x \cdot s_2^z \cdot p_3^y$. By definition of $s_2^y$, $T_3^{y,1}$ reads 2 for $c_{22}^z$. Since $\alpha_3 \cdot s_2^y$ is indistinguishable from $\alpha_3^y$ to $p_3^y$, $T_3^{y,1}$ reads 2 for $c_{22}^z$ in $\alpha_3$. Thus, the write serialization point of $T_2^x$ should come before the read serialization point of $T_4^{x,1}$ in $\alpha_3$ and so before the read serialization points of $T_4^{x,2}, \ldots, T_4^{x,k+1}$. Thus, $T_4^{y,z}$ reads 2 for $c_{22}^z$ in $\alpha_3$.

Similarly, by strict disjoint-access-parallelism, transaction $T_3^{y,1}$ doesn’t access any base object modified in $\beta_3^{y+1}, \ldots, \beta_1^{y+k}$ in $\beta_3^{y+1} \cdot s_2^z$ or in $\gamma_1^y, \ldots, \gamma_k^{y+1}, \gamma_3^{y+1}$. Thus $\alpha_3'$ is indistinguishable from $\alpha_2^x$ to $p_3^y$. By definition of $s_2^y$, transaction $T_3^{y,1}$ reads 0 for $b_{22}^y$ in $\alpha_3^y$. Thus, $T_3^{y,1}$ reads 0 for $b_{22}^y$ in $\alpha_3'$. Then, the write serialization point of $T_2^x$ in $\alpha_3'$ should come after the read serialization point of $T_3^{y,1}$ and so after the read serialization points of $T_3^{y,2}, \ldots, T_3^{y,k+1}$. It follows that $T_3^{y,z}$ reads 0 for $b_{22}^y$ in $\alpha_3'$.

Let $\alpha_4' = \alpha_2^x \cdot s_1^y \cdot \gamma_4^y \cdot \gamma_3^{y+2}$. Since transactions $T_4^{y,1}, \ldots, T_4^{y,k+1}, T_3^{y,1}, \ldots, T_3^{y,k+1}$ do not conflict with each other, by strict disjoint-access-parallelism, it follows that $\alpha_4'$ is legal. Execution $\alpha_3'$ is indistinguishable from $\alpha_4'$ to $p_3^y$ and $p_2^y$. So, it holds that transaction $T_4^{y,z}$ reads 2 for $c_{22}^z$ and $T_3^{y,z}$ reads 0 for $b_{22}^y$ in $\alpha_4'$.

**Step 7. Executions $\beta_5$ and $\gamma_5$ are indistinguishable to process $p_5$.** Consider now the executions $\alpha_5 = \alpha_2 \cdot s_2^z \cdot \beta_3^z \cdot \beta_2^z \cdot \beta_4^z \cdot s_2^y \cdot s_1^y \cdot \gamma_5$, and $\alpha_5' = \alpha_2 \cdot s_2^y \cdot \gamma_5$, $\alpha_2' \cdot s_2^y \cdot \gamma_5$ (Figures 11 and 12). Recall that $\beta_5$ and $\gamma_5$ are solo executions of transaction $T_5$ (which is executed by process $p_5$ and reads the data items $a_1^y, d_2^y, e_3^y$) until $T_5$ commits (obstruction-freedom guarantees that this will occur). Recall also that $\alpha_2^x = \beta_1^x \cdot \beta_2^x \cdot \beta_3^x \cdot \beta_4^x \cdot \beta_5^x$. Since $T_5$ does not conflict with $T_3^{y,z}$ and $T_4^{x,z}$, in executions $\beta_3^z$ and $\beta_4^z$ (and $\gamma_3^z$ and $\gamma_4^z$, respectively), processes $p_3^y$ and $p_2^y$ ($p_3^y$ and $p_3^y$) do not modify any base object read in $\beta_5$ ($\gamma_5$). Moreover, by definition, steps $s_2^z$ and $s_2^z$ do not contend. It follows that $\beta_5$ is indistinguishable from $\gamma_5$ to $p_5$. Thus $T_5$ reads the same values for $a_1^y, d_2^y$, and $e_3^y$ in both $\alpha_5$ and $\alpha_5'$.

**Step 8. Snapshot isolation is violated in either $\alpha_5$ or in $\alpha_5'$.** Recall that $T_3^{y,z}$ reads 1 for $b_{22}^y$ in $\alpha_4$ so it reads 1 for $b_{22}^y$ in $\alpha_5$. Therefore, in $\alpha_5$, the write serialization point of $T_1^y$ precedes the read serialization point of $T_3^{y,z}$, and thus also that of $T_5$. So, $T_5$ reads 1 for $d_2^y$ in $\beta_5$ (notice that no transaction other than $T_1^y$ writes to $d_2^y$). Similarly recall that $T_4^{x,z}$ reads 2 for $c_{22}^y$ in $\alpha_4$ so it reads 2 for $c_{22}^y$ in $\alpha_5'$. Therefore, in $\alpha_5'$, the write serialization point of $T_2^x$ precedes the read serialization point of $T_4^{x,z}$, and thus also that of $T_5$. So, $T_5$ reads 2 for $e_3^y$ in $\alpha_5$ (notice that no transaction other
than $T_2^z$ writes to $e_x$). Since $\beta_5$ is indistinguishable from $\gamma_5$ to $p_5$, $T_5$ reads 1 for $d_y$ and 2 for $e_x$ in both executions. Thus, the write serialization points of $T_1^x$ and $T_2^x$ are placed before the read serialization point of $T_5$ in both executions. Depending on which one of them is going first, $T_5$ reads either 1 or 2 for $a_y^y$. Assume that $T_5$ reads 1 for $a_y^y$ (the case that $T_5$ reads 2 for $a_y^y$ is "symmetric").

We argue that snapshot isolation is violated in $\alpha_5$. Recall that $T_3^{y,z_1}$ reads 1 for $b_{y_1}^y$, in $\alpha_5$, thus the write serialization point of $T_1^x$ must come before the read serialization point of $T_3^{y,z_1}$. $T_3^{y,z_1}$ finishes its execution before the beginning of $T_4^{x,z_2}$, so the read serialization point of $T_3^{y,z_1}$ precedes the read serialization point of $T_4^{x,z_2}$. Recall that $T_4^{x,z_2}$ reads 0 for $c_{x_1}^x$, and therefore also in $\alpha_5$. Thus, the read serialization point of $T_1^{x,z_1}$ must come before the write serialization point of $T_2^z$. It follows that the write serialization point of $T_1^x$ precedes the write serialization point of $T_2^z$, which implies that transaction $T_5$ must read 2 for $a_y^y$. A contradiction.

\[ \square \]

4 SI-DSTM

4.1 Algorithm

We present a simple algorithm, called SI-DSTM, that satisfies obstruction-freedom and R/W-independent snapshot isolation; it also satisfies the additional property [10, 31] that of two concurrent transactions writing to the same data item, only one can commit. The algorithm ensures strict disjoint-access-parallelism between a read-only transaction and any other (read-only or update) transaction. Update transactions are weak disjoint-access-parallel. The algorithm is a simplified version of DSTM [24].

For each active transaction $T$, SI-DSTM maintains a record with fields: (i) $Status$: stores the current status of $T$ (takes values Active, Committed, or Aborted, initially Active), (ii) pendingStatus: records whether $T$ should eventually abort (takes values Active, Committed, or Aborted, initially Committed), and (iii) $readList$: stores information about the data items that are read by $T$.

As in DSTM, for each data item, the algorithm maintains two records, Locator and TMObject (see Algorithm 1). Locator consists of three fields: (1) a pointer to the record of the transaction that holds the ownership of this data item, (2) a copy of its previous value, and (3) a copy of its new value. TMObject contains a reference to a record of type Locator. SI-DSTM ensures a one-to-one correspondence between data items and TMObjects. Thus, when we say that SI-DSTM reads a data item $x$ by calling READTMOBJECT (or WRITEMOBJECT), we mean that READTMOBJECT (or WRITEMOBJECT) is called with a reference to the TMObject that corresponds to $x$ as its argument.

To read a data item $x$, READTMOBJECT finds first the value of $x$ (line 25). If the status of the transaction that holds the ownership of a data item is Active or Aborted, then the value of the object is found in the oldObject field of its locator; otherwise, it is taken from the newObject field of it (see pseudo-code for GETCURRENTVALUE(), lines 16-19). If there is no element for $x$ in $T$’s read list, such an element is added there. Notice that, in contrast to what happens in DSTM, read-only transactions never cause any other transaction to abort. VALIDATEREADLIST() checks whether each data item in $T$’s read list is still consistent (see a discussion related to this at the end of the section).

The comparison of line 22 is performed between references to avoid the ABA problem.

In WRITEMOBJECT(), if the ownership of $x$ is already held by $T$, then the new value is written in the newObject of current $x$’s locator (lines 33-35). Otherwise, as in DSTM, cloning and indirection are employed: a new locator is created for $x$ and its transaction field is initialized to point to the transactional record of $T$ (lines 36-38). Then, $T$ repeatedly tries to change the start field of the TMObject of $x$ to point to this new locator (line 44). Before doing so, it writes the value Aborted in the pending status of the transaction that $T$ found to be the holder of the ownership of $x$.

When $T$ calls COMMITTRANSACTION(), it simply exchanges the value in the pendingStatus field of its transactional record with that of the status field. It then reads status again and returns true or false depending on whether it finds the value Committed or Aborted there.
Algorithm 1: The data structures and pseudo-code of SI-DSTM
Object **ReadTMObject**(Transaction transaction, TMOBJect tmObject):

1. Locator locator = tmObject.start
2. if (locator.transaction == transaction)
3. return locator.newObject
4. Object currentValue = GetCurrentValue(locator)
5. if (not tmObject in transaction.readList)
6. transaction.readList.add(<tmObject, currentValue>)
7. if (not ValidateReadList(transaction))
8. AbortTransaction(transaction)
9. return null
10. return currentValue

Algorithm 2: A modification to ReadTMObject that makes SI-DSTM satisfying snapshot isolation

In SI-DSTM read-only transactions can be invisible. Technically, SI-DSTM does not need maintain shared transactional records for read-only transactions. Each such transaction T needs only to maintain a read list at its private memory space. Moreover, T never causes any other transaction to abort, so it never performs any non-trivial operation to any field of the transactional record of any other transaction. T only reads the status fields of the transactional records of other transactions to discover the current values of the data items read by T and owned by these transactions.

For simplicity, in Algorithm 1, we present a version of SI-DSTM which maintains a transactional record for each transaction including read-only transactions, and in which each read-only transaction performs an exchange operation in CommitTransaction(), however, this action is not necessary for them.

In order to achieve strict disjoint-access-parallelism between any read-only transaction T and update transactions, in SI-DSTM Locator contains an additional field pendingStatus in each transactional record. If a transaction T1 performs a write operation to a data item x for which a transaction T2 holds the ownership, T1 does not write Aborted in the status field of T2; it rather writes this value in pendingStatus to indicate that T2 should abort later. At committing, T2 performs an exchange of pendingStatus and status and finds out that it has to abort. It is only after this exchange that other transactions are aware that T2 aborts. In this way, read-only transactions that read data items owned by T2 contend only with T2 and not T1 or other update transactions that write in the transactional record of T2. However, by the pseudo-code, if T reads the status of T2, then T and T2 conflict on some data item x. Thus strict disjoint-access-parallelism is ensured between a read-only transaction and any other transaction.

Consider three transactions T, T’, and T” and assume that T writes data items x1 and x2, T’ writes x2 and x3, and T” writes x3 and x4. Consider an execution where T, T’, and T” execute sequentially, first T’, then T, and finally T”. Obviously, T and T” do not conflict. Still, they will contend on the status field of T’’s transactional record. Thus, strict disjoint-access-parallelism is not ensured between update transactions. However, it is easy to see that weak disjoint-access-parallelism is ensured even in this case.

Obviously, a transaction will manage to finish its execution successfully, if it runs solo for a sufficient amount of time. However, if a read-only transaction is executed concurrently with update transactions, SI-DSTM does not provide any guarantee that the read-only transaction will not abort repeatedly forever. Moreover, in the presence of contention, an update transaction may never terminate its execution since it may execute the body of the while loop of line 39 forever.

A version of SI-DSTM, that satisfies snapshot isolation instead of R/W-independent snapshot isolation, can be easily derived from the pseudo-code presented in Algorithm 1. In order to do so we need to modify the pseudo-code of the ReadTMObject routine as shown in Algorithm 2. The main difference is that in the new version, a transaction checks whether it holds the ownership to the data item being read; if it is so, the ReadTMObject routine returns the value written by this
4.2 Correctness Proof

For an execution $\alpha$ and a configuration $C$, let $\text{full}(C, \alpha)$ be the sequence of alternating steps from $\alpha$ and configurations resulting from execution of these steps in order starting from $C$.

Recall that for each transaction $T$ there is a unique record of type Transaction and each data item $x$ corresponds to a unique $\text{TMO}bject$. We denote by $\text{trans}_T$ the Transaction object corresponding to transaction $T$.

Fix any execution $\alpha$ of SI-DSTM started from the initial configuration $C_0$. We use the following notation to denote the order relation between two configurations $C_1$ and $C_2$ in $\text{full}(C_0, \alpha)$:

- $C_1 < C_2$ if $C_1$ precedes $C_2$ in $\alpha$;
- $C_1 > C_2$ if $C_2 < C_1$;
- $C_1 \leq C_2$ if either $C_1 < C_2$ or $C_1 = C_2$;
- $C_1 \geq C_2$ if $C_2 \leq C_1$.

We say that a transaction $T$ writes to data item $x$ in $\alpha$ if $T$ executes $\text{WRITE}T\text{MO}bject$ passing a reference to the $\text{TMO}bject$ corresponding to $x$ as an argument. Similarly, $T$ reads the value of data item $x$ in $\alpha$ if $T$ executes $\text{READ}T\text{MO}bject$ passing a reference to the $\text{TMO}bject$ corresponding to $x$ as an argument. Denote by $R(T)$ the set of data items such that $x \in R$ if and only if transaction $T$ reads the value of $x$ in $\alpha$. Similarly, define as $W(T)$ the set of data items such that $x \in W$ if and only if transaction $T$ writes to $x$ in $\alpha$. Without loss of generality, we assume that $R(T) \neq \emptyset$.

For any committed transaction $T$, let $C_w(T)$ be the configuration just after the execution of line 49 in the execution of $T$ in $\alpha$; since $T$ is a committed transaction, the value of the $\text{trans}_T.\text{status}$ field is $\text{Committed}$ at $C_w(T)$. Let $C_r(T)$ be the configuration just after the response of $\text{GET}CURRENT\text{VALUE}$ called at line 25 during the execution of the last instance of $\text{READ}T\text{MO}bject$ executed by $T$.

For any data item $x$, we denote by $tm_x$ the $\text{TMO}bject$ corresponding to $x$. This $\text{TMO}bject$ exists in any configuration $C$ after its creation. We denote by $\text{loc}_x(C)$ the Locator object referenced to by $tm_x.\text{start}$ at $C$.

For any configuration $C$, we define the value of $x$ at $C$, denoted by $v_x(C)$, as follows:

- if $\text{loc}_x(C).\text{transaction}$ is null or $\text{loc}_x(C).\text{transaction}.\text{status}$ is $\text{Committed}$, then $v_x(C) = \text{loc}_x(C).\text{newObject}$;
- otherwise, $v_x(C) = \text{loc}_x(C).\text{oldObject}$.

For any committed transaction $T$ and any $x \in W(T)$, we denote by $nw_{\alpha}(T)$ the value of the third argument of the last execution of $\text{WRITE}T\text{MO}bject$ for $x$ by $T$ in $\alpha$. Informally, $nw_{\alpha}(T)$ is the last value written by $T$ to $x$.

We say that transaction $T$ holds the ownership for data item $x$ in configuration $C$ if $\text{loc}_x(C).\text{transaction}$ points to $\text{trans}_T$. Let $C_1^T(x)$ be the first configuration of $\alpha$ such that $T$ holds the ownership for $x$ at $C_1^T(x)$ and let $C_2^T(x)$ be the first configuration of $\alpha$ such that $C_2^T(x) > C_1^T(x)$ and $T$ doesn’t hold the ownership for $x$ at $C_2^T(x)$. Let also $\alpha_T(x)$ be the execution fragment of $\alpha$ starting from $C_1^T(x)$ up until $C_2^T(x)$.

Let $GCV$ be any instance of $\text{GET}CURRENT\text{VALUE}$ executed by $T$ at line 21 or line 25 in $\alpha$. Denote by $d(GCV)$ a data item defined as follows:

- if $GCV$ is called at line 25 then $d(GCV)$ is the data item corresponding to the $\text{TMO}bject$ passed as the second argument to the instance of $\text{READ}T\text{MO}bject$ which calls $GCV$. 

18
\begin{itemize}
\item if \( GCV \) is called at line 21, then \( d(GCV) \) is the data item corresponding to \( TMObject \) that is accessed by the \textit{for} loop at line 20 just before \( GCV \) was called. \end{itemize}

For convenience of the reader, the used notation is summarized in Figure 13.

\begin{figure}[h]
\centering
\begin{tabular}{|c|l|}
\hline
\( \alpha \) & a fixed execution \\
\( R(T) \) & the read set of a committed transaction \( T \) \\
\( W(T) \) & the write set of a committed transaction \( T \) \\
\( C_r(T) \) & the configuration just after the response of \texttt{GetCurrentValue} called at line 25 during the execution of the last instance of \texttt{READTMOBJECT} executed by \( T \) in \( \alpha \) \\
\( C_w(T) \) & the configuration just after the execution of line 49 in the execution of the last instance of \texttt{READTMOBJECT} by a committed transaction \( T \) \\
\( tm_x \) & the \texttt{TMObject} corresponding to data item \( x \) \\
\( loc_x(C) \) & the Locator object pointed to by \( tm_x.start \) at configuration \( C \) \\
\( v_x(C) \) & if \( loc_x(C).transaction \) is \texttt{null} or \( loc_x(C).transaction.status \) is \texttt{Committed} then \( v_x(C) = loc_x(C).newObject \); otherwise, \( v_x(C) = loc_x(C).oldObject \) \\
\( nv_x(T) \) & the last value written by a committed transaction \( T \) to \( x \) \\
\( C^1_f(x) \) & the first configuration of \( \alpha \) such that \( T \) holds the ownership for \( x \) at \( C^1_f(x) \) \\
\( C^2_f(x) \) & the first configuration of \( \alpha \) such that \( C^2_f(x) > C^1_f(x) \) and \( T \) doesn’t hold the ownership for \( x \) at \( C^2_f(x) \) \\
\( \alpha_T(x) \) & the execution fragment of \( \alpha \) starting from \( C^1_f(x) \) up until \( C^2_f(x) \) \\
\( d(GCV) \) & if \( GCV \) is an instance of \texttt{GetCurrentValue} then \( d(GCV) \) is a data item defined as follows: \\
& - if \( GCV \) is called at line 25, then \( d(GCV) \) is the data item corresponding to \( TMObject \) passed as the second argument to the instance of \texttt{READTMOBJECT} which calls \( GCV \). \\
& - if \( GCV \) is called at line 21, then \( d(GCV) \) is the data item corresponding to \( TMObject \) that is accessed by the \textit{for} loop at line 20 just before \( GCV \) was called. \\
\( GCV_f \) & the instance of \texttt{GetCurrentValue} executed at line 21 in the \textit{last} call of \texttt{READTMOBJECT} for the fixed data item \( x \) by the fixed transaction \( T \) \\
\( GCV_i \) & the instance of \texttt{GetCurrentValue} executed at line 21 in the \textit{last} call of \texttt{VALIDATEREADLIST} for the fixed transaction \( T \) \\
\( C_f \) & a configuration such that \( GCV_f \) returns \( v_x(C_f) \), and this configuration is between the read of the parameter \( GCV_f \) and its return \\
\( C_l \) & a configuration such that \( GCV_i \) returns \( v_x(C_l) \), and this configuration is between the read of the parameter \( GCV_i \) and its return \\
\hline
\end{tabular}
\caption{Notation used in this section}
\end{figure}

\textbf{Theorem 4.1.} SI-DSTM satisfies R/W-independent snapshot isolation.

Informally, the outline of the proof is the following. First, in Lemma 4.2, we prove that transaction \( T \) holds the ownership for all data items in its write set \( W(T) \) in configuration \( C_w(T) \). This implies that for all \( x \in W \), \( v_x(C_w) = nv_x(T) \). Then, in Lemma 4.3, we prove that for any configuration \( C \) and data item \( x \), \( v_x(C) \) is the value written by the last committed transaction which executed line 49. These two lemmas imply that the write serialization point for \( T \) must be placed at \( C_w(T) \).

We also provide two lemmas to prove that the read serialization point for \( T \) can be placed at \( C_r(T) \). In Lemma 4.4, we prove that, for any instance \( GCV \) of \texttt{GetCurrentValue} there is a configuration \( C \) between the invocation of \( GCV \) and its response such that \( GCV \) returns the value of \( d(GCV) \) at \( C \). In other words, we ensure that \( GCV \) doesn’t return a value that this data item had at some obsolete configuration. Lemma 4.4 is used for proving Lemma 4.5 which states that for every data item \( x \) and every pair of configurations \( C \) and \( C' \) after the first read of \( x \) and before the last validation, \( v_x(C) = v_x(C') \). We argue that \( C_r(T) \) satisfies the constraints of Lemma 4.4.
and Lemma 4.5. This allows us to prove that by placing the read serialization point of $T$ at $C_r(T)$, SI-DSTM ensures R/W-independent snapshot isolation.

**Lemma 4.2.** Let $T$ be a committed transaction in execution $\alpha$. Then for any $x \in W(T)$, $C_w(T) \in \alpha_T(x)$.

**Proof.** Obviously, $C_w(T) > C_1^T(x)$ because transaction $T$ commits just after executing line 49 (i.e. after $C_w(T)$) and so it cannot call any other routines after $C_w(T)$. Assume by contradiction, that $C_w(T) > C_2^T(x)$.

Let $C$ be the configuration just before $C_2^T(x)$ in $\text{full}(C_0, \alpha)$. Since, at $C$, transaction $T$ still holds the ownership of $x$, $\text{loc}_x(C).\text{transaction} = \text{trans}_T$. By the pseudo-code, there exists a transaction $T'$ such that $x \in W(T')$ and $T'$ executes successfully the CAS operation at line 44 at $C$. Thus, it follows that $T'$ read $\text{loc}_x(C)$ when it read $\text{tnObject}.\text{start}$ either at line 32 or on line 46. Thus, $T'$ read $\text{trans}_T$ in oldLocator.\text{transaction} (line 40), evaluated the if condition of line 41 to \textbf{true} and changed the pendingStatus of $\text{trans}_T$ to Aborted at line 42. Since $C_w(T) > C_2^T(x)$ and pendingStatus is changed to Aborted before $C_2^T(x)$, it follows that $T$ aborts in $\alpha$, which is a contradiction. So, $C_w(T) \leq C_2^T(x)$ and it holds that $C_w(T) \in \alpha_T(x)$. \hfill $\Box$

**Lemma 4.3.** Let $T$ be a committed transaction and $x \in W(T)$ be any data item. For each configuration $C \geq C_w(T)$ in $\text{full}(C_0, \alpha)$, if there is no committed transaction $T'$ such that $x \in W(T')$ and $C_w(T) < C_w(T') \leq C$, then $v_x(C) = n_x(T)$.

**Proof.** Assume by contradiction that the claim is not true and there is at least one configuration $C''$ such that $C_w(T) < C'' \leq C$ and $v_x(C'') \neq n_x(T)$. Let $C'$ be the first such configuration. Lemma 4.2 implies that $v_x(C_w(T)) = n_x(T)$, then $v_x(C) \neq v_x(C_w(T))$

Let $C^-$ be the configuration just before $C'$ in $\text{full}(C_0, \alpha)$. As $C^-$ and $C'$ are consecutive configurations and $v_x(C^-) \neq v_x(C')$, then, by the pseudo-code, it holds that there exists a committed transaction $T'$ such that $C' = C_w(T')$ and $T'$ holds the ownership of $x$ at $C'$. We reach a contradiction. \hfill $\Box$

**Lemma 4.4.** Let $\text{GCV}$ be any instance of GetCurrentValue, let $x = d(\text{GCV})$ and $r$ be the read of $\text{tm}_{\alpha}.\text{start}$ performed by $T$ in order to determine the argument of $\text{GCV}$. Let $C_1$ be the configuration just before $r$ is performed and $C_2$ be the configuration just after $\text{GCV}$ responds. Then, there exists a configuration $C$, such that $C_1 \leq C \leq C_2$ and $\text{GCV}$ returns $v_x(C)$.

**Proof.** The argument passed to $\text{GCV}$ is a reference to $\text{loc}_x(C_1)$. Notice that any Locator object is immutable (i.e. its fields do not change their values) after the execution of line 14 or the successful execution of CAS at line 44.

If $\text{loc}_x(C_1).\text{transaction} = \text{null}$ or $\text{loc}_x(C_1).\text{transaction} = \text{trans}_T$, then let $C = C_1$. We argue that the claim holds. In the first case, by the pseudo-code (lines 17-18) and by the definition of $\nu(C_1)$, $\text{GCV}$ returns $v_x(C) = v_x(C_1) = \text{loc}_x(C_1).\text{newObject}$. In the later case, as $\text{trans}_T.\text{status} = \text{Active}$ and $T$ doesn’t change $\text{trans}_T.\text{status}$ in the execution of $\text{GCV}$, then, by the pseudo-code (lines 17-19), $v_x(C) = v_x(C_1) = \text{loc}_x(C_1).\text{oldObject}$.

Now assume that $\text{loc}_x(C_1).\text{transaction} = \text{trans}_T$, where $T' \neq T$ is some transaction. Let $C'$ be the configuration in $\text{full}(C_0, \alpha)$ just before $\text{GCV}$ reads $\text{trans}_T.\text{status}$ at line 17.

If $\text{trans}_T.\text{status}$ is Aborted or Active on $C'$, then let $C = C_1$. By the pseudo-code, $\text{GCV}$ returns $\text{loc}_x(C_1).\text{oldObject}$. Obviously, $\text{trans}_T.\text{status}$ cannot be Committed at $C_1$, thus, by the definition of $v_x(C)$, $v_x(C_1) = \text{loc}_x(C_1).\text{oldObject}$, and $\text{GCV}$ return $v_x(C)$.

If $\text{trans}_T.\text{status}$ is Committed at $C'$, then, by the pseudo-code, $\text{GCV}$ returns $\text{loc}_x(C_1).\text{newObject}$. We consider two cases. Assume first that $\text{trans}_T.\text{status}$ is Active at $C_1$. Then, it holds that $C_1 < C_w(T') \leq C'$. By the definition, $v_x(C_w(T')) = \text{loc}_x(C_w(T')).\text{newObject}$. Let $C = C_w(T')$ in this case.
If \( trans_T.\text{status} \) is Committed at \( C_1 \), let again \( C = C_1 \). Then \( v_x(C_1) = \text{loc}_x(C_1).\text{newObject} \) and the claim holds when \( C = C_1 \).

Thus, in both cases GCV returns \( v_x(C) \).

Fix any committed transaction \( T \) and any data item \( x \in R(T) \). Let \( \text{GCV}_f \) be the instance of \( \text{GetCurrentValue} \) executed at line 25 in the first call of \( \text{ReadTMObject} \) for \( x \) by \( T \), and let \( \text{GCV}_V \) be the instance of \( \text{GetCurrentValue} \) executed at line 21 in the last call of \( \text{ValidateReadList} \) by \( T \). Lemma 4.4 implies that there exists a configuration \( C_f \) (\( C_l \)) such that \( \text{GCV}_f \) (\( \text{GCV}_V \), respectively) return \( v_x(C_f) \) (\( v_x(C_l) \), respectively) and this configuration is between the read of the parameter \( \text{GCV}_f \) (\( \text{GCV}_V \), respectively) and its return.

**Lemma 4.5.** For any configuration \( C \) such that \( C_f \leq C \leq C_l \), \( v_x(C) = v_x(C_f) = v_x(C_l) \).

**Proof.** Recall that \( trans_T \) contains the \( \text{readList} \) field which stores the read list of \( T \). Notice that \( \text{readList} \) can be modified by \( T \) only at line 27. The check of the if statement at line 26 ensures that only the first call of \( \text{ReadTMObject} \) for each data item modifies \( \text{readList} \). So the data item together with the value of \( x \) at \( C_f \) is stored in \( \text{readList} \) by executing line 27. This value is later compared with \( v_x(C_f') \) by executing line 22 in \( \text{ValidateReadList} \). Since \( T \) commits, it holds that \( v_x(C_f) = v_x(C_l) \).

Assume by contradiction that there exists at least one configuration \( C'' \) such that \( C_f < C'' < C_l \) and \( v(C'') \neq v(C_f) \). Denote the first such configuration by \( C' \) and let \( C'^{-} \) be its preceding configuration. As \( C^{-} \) and \( C' \) are consecutive configurations and \( v_x(C^{-}) \neq v_x(C') \), by the pseudo-code, it follows that there exists a committed transaction \( T' \) such that \( C' = \text{W}_w(T') \). By the pseudo-code of the algorithm, \( v(C'^-) = \text{loc}(C'^-).\text{oldObject} \), \( v_x(C') = \text{loc}_x(C).\text{newObject} \) and the value of \( x \) is never equal to \( \text{loc}_x(C'^-).\text{oldObject} \) at any configuration after \( C' \).

Because of cloning at lines 34 and 38 all values of \( x \) are unique (the values are the same only if they have equal references), thus it follows that \( v_x(C_l) \neq v_x(C_f) \). We reach a contradiction.

By the definition of \( \text{C}_r(T) \), \( \text{C}_r(T) \) occurs before the last call of \( \text{ValidateReadList} \) at line 28 by \( T \). Since \( trans_T.\text{readList} \neq \emptyset \), \( \text{GCV}_V \) is an instance of \( \text{GetCurrentValue} \) called by this \( \text{ValidateReadList} \). By definition of \( C_l \), it follows that \( \text{C}_r(T) < C_l \). Obviously, \( \text{C}_r(T) > C_f \) by definitions of \( \text{C}_r(T) \) and \( C_f \). Thus, \( C_f \leq \text{C}_r(T) \leq C_l \), and Lemma 4.5 implies the following corollary:

**Corollary 4.6.** If \( T \) is a committed transaction then, for every \( x \in R(T) \), each \( \text{ReadTMObject} \) performed by \( T \) and taking \( \text{tm}_x \) as the argument returned value of \( x \) at \( \text{C}_r(T) \).

Let \( T' \) be the committed transaction such that \( x \in \text{W}(T') \) and the write serialization point of \( T' \) is the last write serialization point before the read serialization point of \( T \). By the way isolation points are assigned, \((T')_w \) is linearized at \( \text{W}_w(T') \). By the definition of \( T' \), there is no committed transaction \( T'' \) such that \( x \in \text{W}(T'') \) and \( \text{W}_w(T') < \text{W}_w(T'') < \text{C}_r(T) \). Thus, Lemma 1.3 implies that \( v_x(\text{C}_r(T)) = \text{nv}_x(T') \) which in turn implies that R/W-independent snapshot isolation is ensured.

### 4.3 Proof of Obstruction-Freedom and Disjoint-Access-Parallelism

We continue to prove that SI-DSTM satisfies obstruction-freedom.

**Theorem 4.7.** SI-DSTM is obstruction-free.

**Proof.** Let \( \alpha \) be any execution and \( T \) be any transaction in this execution. Recall that a TM algorithm is obstruction-free if a transaction \( T \) can be aborted only when processes other than the one executing \( T \) take steps during the execution interval of \( T \).

Assume that no other process takes steps during the execution of \( T \). By the pseudo-code, \( T \) can be aborted (i.e. \( trans_T.\text{status} \) set to \text{Aborted} only in two cases:

- the \( \text{pendingStatus} \) field of its transactional record is set to \text{Aborted} before the execution of line 49 by \( T \);
• an instance of ValidateReadList executed by \( T \) returns \( \text{false} \).

The first case is not possible, because \( T \) sets pendingStatus to \text{Committed} on line 6 and never modifies its value afterwards, so, given that no other process has taken steps during the execution of \( T \), when \( T \) executes the exchange on line 49 it writes \text{Committed} into the status field of its transactional record.

Assume now that an instance \( \text{VRL} \) of ValidateReadList called by \( T \) returns \( \text{false} \) (line 23) and let \( x \) be the data item corresponding to \text{tmObject} that is accessed by the for loop of line 20. Under the assumption that no other transaction is running concurrently to \( T \), the check on line 22 executed during \( \text{VRL} \) cannot fail. This is so, since by the pseudo-code, values written by \( T \) cannot be returned as the result of an execution of any \text{GETCURRENTVALUE} instance before \( T \) commits. So we reach a contradiction, thus the claim holds.

Clearly, SI-DSTM doesn’t satisfy strict disjoint-access-parallelism between update transactions. To prove this consider a counterexample execution with three transactions: \( T_1 \) which writes to data items \( x \) and \( y \), \( T_2 \) which writes to \( x \), and \( T_3 \) which writes to \( y \). If \( T_1 \) performed writes to \( x \) and \( y \) but not committed yet, and then \( T_2 \) and \( T_3 \) perform writes to \( x \) and \( y \), respectively, then both \( T_2 \) and \( T_3 \) may contend on \text{trans}_{T_1}.pendingStatus when executing line 42. Thus, \( T_2 \) and \( T_3 \) contend on the same base object while they do not conflict.

We will define a new form of disjoint-access-parallelism, called read-disjoint-access-parallelism, and we will show that SI-DSTM satisfies this property.

**Definition 4.8.** We say that a TM implementation is read-disjoint-access-parallel, if, for each execution \( \alpha \) and every two transactions \( T_1 \) and \( T_2 \) in \( \alpha \), if \( \alpha | T_1 \) and \( \alpha | T_2 \) contend on some base object, then one of the following conditions holds:

- both \( T_1 \) and \( T_2 \) are update transactions and there is a path between \( T_1 \) and \( T_2 \) in the conflict graph of the minimal execution interval of \( \alpha \) containing \( \alpha | T_1 \) and \( \alpha | T_2 \);
- at least one of \( T_1 \) and \( T_2 \) is a read-only transaction and there is an edge between \( T_1 \) and \( T_2 \) in the conflict graph of \( \alpha \).

Informally, read-disjoint-access-parallelism states that weak disjoint-access-parallelism is ensured between update transactions and strict disjoint-access-parallelism is ensured between a read-only transaction and all other transactions.

Finally, we provide the proof that SI-DSTM satisfies read-disjoint-access-parallelism.

**Theorem 4.9.** SI-DSTM is read-disjoint-access-parallel.

**Proof.** Let \( \alpha \) be any execution. By the pseudo-code, any Locator record is immutable (i.e. its fields do not change their values) after the execution of line 14 or the successful execution of CAS on line 44. Thus, no two transactions can contend on any field of a Locator record. Also by the pseudo-code, for any transaction \( T \) in \( \alpha \), \text{trans}_{T}.readList can be accessed by transaction \( T \) only, thus no transaction can contend with \( T \) on \text{trans}_{T}.readList.

Let \( T_1 \) be any read-only transaction executed in \( \alpha \). By the pseudo-code, a read-only transaction never acquires the ownership of any data item. Hence, the transaction field of any Locator record is not a reference to the transactional record of a read-only transaction. This means that no other transaction can read or modify any field of \text{trans}_{T_1}, so \( T_1 \) and any other transaction cannot contend on \text{trans}_{T_1}.status or \text{trans}_{T_1}.pendingStatus.

Assume that \( T_1 \) stores a non-null value that is a reference to the transactional record of a transaction \( T_2 \) to its local variable \text{currentTransaction} by executing line 16. By the pseudo-code, \( T_2 \) is an update transaction and \( T_1 \) and \( T_2 \) conflict on the same data item. It follows that there is an edge between \( T_1 \) and \( T_2 \) in the conflict graph of \( \alpha \).
Assume now that $T_1$ and $T_2$ are two update transactions that contend on some base object $o$ and at least one of them modifies the value of $o$ in $\alpha$. Without loss of generality, let $T_1$ be this transaction. By inspecting the pseudo-code, we consider the following cases:

1. $o$ is $\text{trans}_T1.status$ and $T_1$ contend with $T_2$ when $T_1$ executes line 49. It follows that $T_2$ reads $\text{trans}_T1.status$ on line 17 (notice that this is the only line in the pseudo-code that a transaction reads the status field of a transactional record of some other transaction). Thus $T_1$ and $T_2$ conflict on the same data item $x$; specifically, $T_1$ holds the ownership of $x$ and $T_2$ reads its value;

2. $o$ is $\text{trans}_T1.pendingStatus$ and $T_1$ contend with $T_2$ when $T_1$ executes line 49. It follows that $T_2$ has modified $\text{trans}_T1.pendingStatus$ by executing line 42. Again, $T_1$ and $T_2$ conflict on the same data item $x$, specifically, both $T_1$ and $T_2$ write to $x$;

3. $o$ is $\text{trans}_T2.pendingStatus$ and $T_1$ contend with $T_2$ when $T_1$ executes line 42. This case is symmetric to the previous one;

4. $o$ is $\text{trans}_T3.pendingStatus$, where $T_3$ is some transaction, other than $T_1$ and $T_3$, and $T_1$ contend with $T_3$ when $T_1$ executes line 42. By the assumption, $T_1$ and $T_2$ contend on $\text{trans}_T3.pendingStatus$. Since no transaction ever reads the pendingStatus field of any transactional record, it must be that $T_2$ also executes line 42 and contend with $T_3$ on $\text{trans}_T3.pendingStatus$. It follows that $T_1$ and $T_3$ conflict, $T_2$ and $T_3$ conflict and thus there is a path between $T_1$ and $T_2$ in the conflict graph of the minimal execution interval of $\alpha$ containing $\alpha | T_1$ and $\alpha | T_2$;

5. $o$ is $tm_{x}.start$ where $x$ is some data item and $T_1$ contend with $T_2$ when $T_1$ executes line 44. By the pseudo-code, $T_2$ also executes line 44 when $T_2$ writes to $x$, so $T_1$ and $T_2$ conflict.

□
References


