# Block Transactional Memory: A Complexity Study

**Supra Research**

24 February 2025

**Abstract**

In the smart contract ecosystem, the transaction contains the code to be executed, and all nodes that execute the block of transactions must arrive at the same final state. Consequently, it becomes necessary to maximize the parallel execution of transactions within a node and still allow all nodes to reach a consistent final state resulting from the execution of a set of transactions.

Traditional software transactional memory has been identified as a possible abstraction for concurrent executions of transactions within a block. However, in the smart contract setting, the execution order must follow the preset order of the transactions in the block. This paper presents a complexity study of this model for in-memory transactions (or block transactional memory) and investigates the fundamental lower bounds that might exist. We ask the question: Do we need to maintain multiple versions of the values associated with each account? To answer this, we formalize the model of block transactional memory and identify the safety property that is needed. We show that, under natural restrictions of liveness, it is necessary for smart contract transactions to maintain a large number of versions for each account or allow read-only transactions to write to shared memory. We then present several algorithmic designs for single-version and multi-version block transactional implementations that provide preset serializability.

## 1 Introduction

For higher throughput, smart contract ecosystems rely on parallel execution of a *block* of user-specified *transactions* that each perform a sequence of *reads* and *writes* on user accounts [6, 13, 14, 15, 19, 30]. A *block proposer* node takes as input a *block* consisting of $n$ transactions and a *preset* total order $T_1 \to T_2 \ldots, \to T_n$. The node attempts to execute, in parallel, the $n$ transactions such that the state resulting from the concurrent execution of the $n$ transactions must be the state resulting from the *sequential* execution of transactions $T_1 \cdot T_2 \cdots T_n$.

As with traditional shared memory algorithms, we may consider *coarse-grained* or *fine-grained locks* or *atomic transactions* for concurrent executions of the user-specified transactions. While safety for traditional shared memory allows the concurrent execution to be *equivalent* to *some* sequential execution (known as *serializability*), safety for smart contract transactions requires the concurrent execution must be equivalent to the exact sequential execution that respects the preset order (transaction order) of the user-specified transactions in the block.

This paper studies the complexity of *transactional memory (TM)* algorithms for safe parallel execution of smart contract transactions [15]. Intuitively, if the execution must respect the preset order, it seemingly restricts concurrency significantly. For instance, if we consider that transaction $T_1$ must precede $T_2$ in the preset order (denoted $T_1 \to T_2$), without *a priori* knowledge of the read-write conflicts on the set of accounts accessed, committing $T_2$ prior to the commit of $T_1$ may result in a safety violation. To illustrate this, consider the following execution: $T_1$ performs the read of an account $X_1$ (reading value 0), following which $T_2$ performs a read of an account $X_2$ (reading value 0), followed by a write of a new value 1 to account $X_1$, as shown in Figure 1. Observe in Figure 1a that if $T_2$ commits at this point in the execution and if $T_1$ may wish to perform a write of a new value 1 to $X_2$ after the commit of $T_2$, then the resulting execution does not respect the preset order in any extension. Clearly, if transaction $T_1$ commits, the resulting execution is not equivalent to any sequential execution. Alternatively, if $T_1$ aborts and then re-starts the execution, any read of the account $X_2$ will return the value 1 that is written by $T_2$, thus not respecting the preset order, as illustrated in Figure 1b. Consequently, Figure 1c shows that the only possible mechanism to avoid

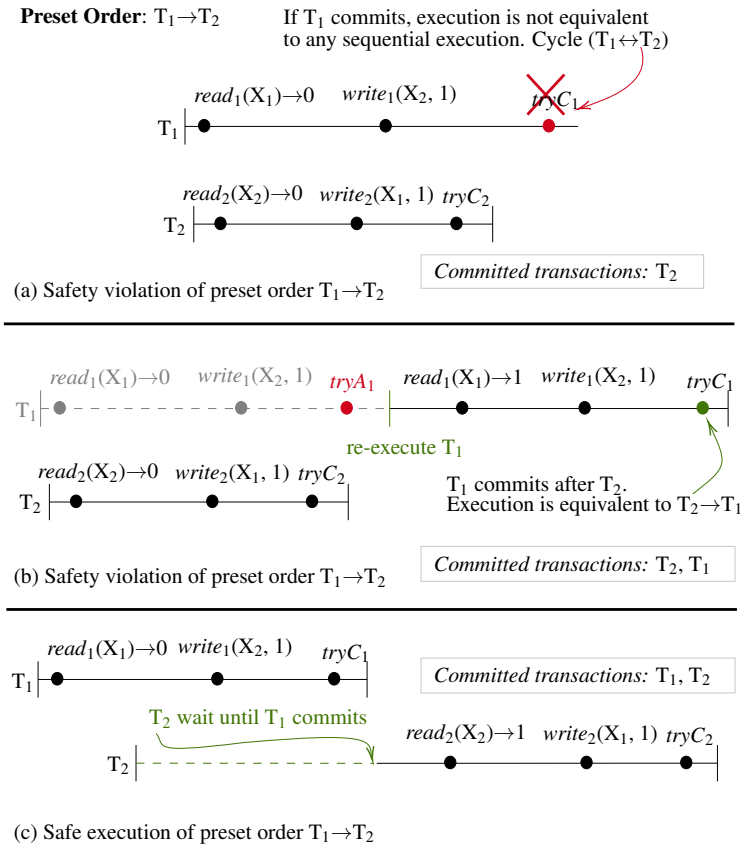


**Figure 1** Safe execution of transactions in preset order: sub-figure (a) illustrates that when conflicting transactions $T_1$ and $T_2$ execute and commit in an arbitrary order, it will result in an unsafe execution; (b) illustrates that transactions commit in some serialization order other than preset serialization, resulting in an unsafe execution. In sub-figure (c), to ensure safe execution, $T_2$ waits for $T_1$ to commit before committing.

this requires $T_2$ to wait until $T_1$ commits. This observation guides the motivation for this paper and to undertake the study of the complexity of block transactional memories.

## Contributions

In this paper, we formalize the model of *block transactional memory* [15] (à la traditional *transactional memory (TM)* [26]). While traditional TMs adopt *serializability and its variants for safety*, block TMs need the serialization to respect a pre-specified preset order. We precisely define this property of preset serializability and ask if block transactional memory implementations that satisfy preset serializability necessarily have a significant space complexity. More specifically, given a set of $n$ transactions and a preset order $T_1 \rightarrow T_2 \ldots, \rightarrow T_n$, what is the cost of concurrency in terms of the number of versions of each account that must be maintained? Our main lower bound shows that, under natural restrictions of liveness, it is necessary for transactions that enforce preset serializability in block transactional memory implementations to maintain a large number of versions for each account or allow read-only transactions to write to shared memory. We then present the design of single-version and multi-version block transactional implementations that provides preset serializability.

### Roadmap

The rest of this paper is structured as follows: Section 2 presents a discussion of related techniques for smart contract parallelization and relationship of block transactional memory to traditional transactional memory algorithms. In Section 3, we presents the technical preliminaries needed to define the safety properties and characterize the complexity of block transactional memories. While, Section 3.2 presents our formalization of the block transactional memory model and identify how they are related to each other. Section 4 presents the family of block transactional memory algorithms and how they can be augmented with hardware transactional support prevalent in today's CPU architectures. Section 5.3 derives the construction of the main lower bound in the paper for the space complexity of block transactional memories. We conclude with a discussion of the conceptual takeaways from this paper and the empirical performance of block transactional memory implementations in today's multi-socket shared memory architectures.

## 2 Related work

In this section, we provide a detailed background on the different smart contract execution models, existing algorithms for block transactional memory and how traditional transactional memory model and results are related to those presented in this paper.

### 2.1 Models for parallel execution of smart contracts

There have been several models of parallel execution of smart contract transactions in the blockchain setting. In one model, the *block proposer* executes the transactions, creates the block with state-differences and dependency information, and then propagates the block in the network for validators to validate. In another model, the block proposer proposes the order of transactions in the block, and all the nodes come to an agreement on the order and then execute the transactions parallelly in the order. The first model is called the *Ethereum model* [12], while the second one we call the *Aptos model* [5]. Similarly, when transactions consist of read and write set information for parallel execution, we call that model the *Solana model* [27]. There is another model where resource ownership is exploited for parallel execution; we call this model the *Sui model* [29].

The above four models of execution can be classified into two distinct classes for speeding up execution: The first class consists of techniques that employ runtime execution, such as transactional memory, and execute the transaction speculatively in parallel; we call it *read-write oblivious* execution. The second class, called *read-write aware* execution, requires the client's transaction access hints to execute transactions in parallel through preprocessing or runtime techniques.

#### Read-write oblivious execution (runtime speculative techniques).

In this class of executor, the goal is to execute an ordered set of transactions in parallel as if they were executed sequentially and arrive at the same final state. The key idea is that some transactions may not be conflicting, namely, they do not read or write any common data items; therefore, they can be processed in parallel, enabling execution acceleration that arrives at the correct sequential result.

For the Ethereum model, the first pioneering work on parallel execution of block transactions is proposed by Dickerson et al. [11]. They proposed to use block transactional memory (pessimistic ScalaSTM) for parallel execution of transactions at the block proposer and a happen-before graph (a directed acyclic graph appended to the block by the proposer) for parallel execution at the validators. Later, Anjana et al. [3, 4] proposed an optimistic-STM based multi-version concurrency control (multi-version timestamp ordering protocol) for parallel execution at the block proposer and directed

acyclic graph based efficient and safe parallel execution at validators. Saraph and Herlihy [23] proposed a simple *bin-based two-phase* approach. In the first phase, the proposer uses locks and tries to execute transactions concurrently by rolling back those that lead to conflict(s). All the aborted transactions are kept in a sequential bin and executed sequentially in the second phase. The proposer gives concurrent and sequential bin hints in the block to the validator to execute the same schedule as executed by the proposer.

### Read-write oblivious execution with block transactional memory.

In Aptos model of parallel execution, post-ordering acceleration of transaction based on preset serializable and multi-version concurrency control is proposed in the Block-STM [15]. Rather than speculatively executing block transaction in any order, they employ it on ordered-sets, called the preset serialization order. Each validator uses Block-STM independently to execute a leader proposal of ordered set of transactions in parallel to get the same final state. This is currently been implemented in the Aptos blockchain [5] and is the most promising approach, since it does not require any additional information to be appended to the transactions or in the block for parallel execution.

### Read-write aware execution (techniques based on hints by clients).

Transactions on blockchains like Solana [27] and Hyperledger Sawtooth [24] come up with the specification of accounts they access in read and write mode during execution. These access hints are utilized by the block proposers, in which the proposer statically analyses transactions to execute them in parallel or employs a runtime scheduler that schedules the transaction to execute in parallel based on read-write set conflicts and finally proposes the block, including the hints for parallel execution at validators. The validators can execute the block transactions in parallel with the block proposer's hint, which is a crucial component of this approach. In another approach, preprocessing or static analysis of read-write sets can also be used to generate a dependency graph between transactions. The graph can then be used to split the transactions into different groups and execute them in parallel. Moreover, read-write transaction information can be used as a seed for parallel execution in Block-STM [15] and other approaches, as discussed in the previous section, to reduce abort rates.

For the Ethereum model, Amiri et al. [2] proposed *ParBlockchain*, a technique for parallelly executing block transactions in a permissioned blockchain using static analysis or speculative execution to obtain the read-set and write-set of each transaction, then generates the dependency graph and constructs the block. Their approach adds an extra overhead of preprocessing for parallel execution. SeaLevel executor [30, 31] implemented in Solana [27] uses the transaction read-write set information and locks to execute transactions in multiple iterations at the block proposer. There are two phases involved in every iteration: the locking phase and the execution phase. Using read-write set information, lock profiles are generated for transactions in the locking phase. Those transactions that are not in conflict with others can be executed in parallel during the execution phase of the iteration, while those that are in conflict are passed on to the following iteration until all the transactions in the block are executed. The block proposer provides the iteration information inside the block for parallel execution at validators. Similarly, Hyperledger Sawtooth [24] developed a tree-based scheduler for parallel execution utilizing the read-write set information with transactions. Instead of using a tree for parallel execution in Sawtooth, an efficient concurrent DAG-based parallel scheduler has been proposed in [22].

Sui [29] defines a state storage model (called the object model) that facilitates the identification of independent transactions. The objects may be shared or have exclusive ownership. Each object is assigned a unique identity (ID) and consists of references to the owners' addresses, allowing multiple users or transactions to access them (read or write). Using the object ownership model, it is easy

to identify dependencies by checking if transactions access the same object. Based on this object ownership model, Sui [29] introduced parallel execution of transactions [13, 14], where transactions that interact with owned objects can be executed in parallel because no other transaction will conflict with those objects, while transactions that access shared objects are executed sequentially since they may lead to conflicting accesses. Moreover, transactions that do not involve a shared object can completely bypass consensus.

## 2.2 Traditional transactional memory and block transactional memory

As mentioned earlier, Aptos's Block-STM [15] is the most recent block transactional memory implementation that requires the execution of a set of transactions to be equivalent to a fixed preset serialization order, which restricts concurrency significantly. Block-STM simplifies the concurrent execution requirement of sharing transaction dependency information within the block by the block proposer for validators to get the same final state because each node is conforming to the presetexecution order.

In this design, there are two tasks for each transaction: the execution task and the validation task, prioritizing tasks for transactions lower in the serialization order $S$. A transaction may be executed and validated several times by the *scheduler*. For each incarnation $i$ of a transaction $T_k$, Block-STM maintains two transaction local buffers: a read-set ($Rset_k^i$) and a write-set ($Wset_k^i$). The account and respective versions that are read during the execution of the incarnation are contained in the $Rset_k^i$. The update is represented by (account, value) pairs and is stored in the $Wset_k^i$. For an incarnation $i$ of $T_k$ the $Wset_k^i$ is stored in the multi-version in-memory data structure. Further, an incarnation of a transaction must pass validation once it executes. The validation compares the observed versions after re-reading (double-collecting) the $Rset_k^i$. Intuitively, a successful validation indicates that the $Wset_k^i$ of an incarnation $i$ of $T_k$ is still legitimate and all reads in the $Rset_k^i$ are consistent, while an unsuccessful validation implies that the incarnation $i$ must be aborted. When a transaction $T_k$ is aborted, it implies that all higher transactions in $S$ than $T_k$ can be committed only if they get successfully validated afterwards.

Block-STM conforms to our lower bound presented in this paper (cf. Theorem 6). The formalization in this paper and whether the "ordering curse" from the presetordering necessitates multiple versions is inspired by the design and presentation outlined in [15].

### Relationship between traditional transactional memory and block transactional memory.

A traditional TM implementation guarantees that the concurrent execution of a set of transactions is equivalent to some sequential execution. The implementation proposed in [4, 11] follows this approach at block proposer and requires block proposer to share the execution schedule in the form of a directed acyclic graph with validators to reach the same final state.

At a conceptual level though, an implementation of transactional memory and state-of-the-art algorithms like TL2 [10], NOrec [9], DSTM [25], etc. will not be useful for Block-STM nor will the algorithmic techniques they employ nor any complexity bounds applicable here. That said, there are proof techniques that have been employed for deriving complexity bounds in traditional TMs that are very much applicable in the Block-STM context which we leverage in this paper.

Perelman et al. [21] proved that a large class of TMs called *mv-permissive* TMs (a transaction can only be aborted if it is an updating transaction that read-write conflicts with another updating transaction) cannot be *online space optimal*, *i.e.*, no mv-permissive TM can keep the minimum number of old object versions for any execution. Similarly, Kuznetsov et al. [18] showed that TMs providing *wait-free* read-only transactions must maintain all the versions of the writes performed by

updating transactions. The formalization in Section 5.3 is based on [18] and the proof of Theorem 6 is based on analogous results in [18, 21].

# 3 Transactional memory: Model and definitions

In this section, we formally define the model and properties of block transactional memory. We present this as an extension of the formalism associated with models of traditional transactional memory [16]. Throughout this paper, we consider the setting where a *block* of $n$ smart contract transactions have to be executed concurrently on an asynchronous shared memory machine.

## 3.1 Definitions and technical preliminaries

### Transactional memory (TM)

A *transaction* is a sequence of *transactional operations* (or *t-operations*), reads and writes, performed on a set of *objects* (alternatively, we can consider an object to represent a *user account*). A TM *implementation* provides a set of concurrent *processes* with deterministic algorithms that implement reads and writes on accounts using a set of *shared memory locations*. More precisely, for each transaction $T_k$, a TM implementation must support the following t-operations: $read_k(X)$, where $X$ is an account, that returns a value in a domain $V$ or a special value $A_k \notin V$ (*abort*), $write_k(X, v)$, for a value $v \in V$, that returns *ok* or $A_k$, and $tryC_k$ that returns $C_k \notin V$ (*commit*) or $A_k$. The transaction $T_k$ completes when any of its operations return $A_k$ or $C_k$.

### Configurations and executions.

A *configuration* of a TM implementation specifies the state of each location and each process. In the *initial* configuration, each location has its initial value and each process is in its initial state. An *event* (or *step*) of a transaction invoked by some process is an invocation of a t-operation, a response of a t-operation, or an atomic *primitive* operation applied to a location along with its response. An *execution fragment* is a (finite or infinite) sequence of events $E = e_1, e_2, \ldots$. An *execution* of a TM implementation $M$ is an execution fragment where, informally, each event respects the specification of shared memory locations and the algorithms specified by $M$.

For any finite execution $E$ and execution fragment $E'$, $E \cdot E'$ denotes the concatenation of $E$ and $E'$, and we say that $E \cdot E'$ is an *extension* of $E$. For every transaction identifier $k$, $E|k$ denotes the subsequence of $E$ restricted to events of transaction $T_k$. If $E|k$ is non-empty, we say that $T_k$ *participates* in $E$, Let $txns(E)$ denote the set of transactions that participate in $E$. Two executions $E$ and $E'$ are *indistinguishable* to a set $\mathscr{T}$ of transactions, if for each transaction $T_k \in \mathscr{T}$, $E|k = E'|k$. A transaction $T_k \in txns(E)$ is *complete in* $E$ if $E|k$ ends with a response event. The execution $E$ is *complete* if all transactions in $txns(E)$ are complete in $E$. A transaction $T_k \in txns(E)$ is *t-complete* if $E|k$ ends with $A_k$ or $C_k$; otherwise, $T_k$ is *t-incomplete*.

Two executions $E$ and $E'$ are *indistinguishable* to a set $\mathscr{T}$ of transactions, if for each transaction $T_k \in \mathscr{T}$, $E|k = E'|k$.

### History.

A TM *history* is the subsequence of an execution consisting of the invocation and response events of t-operations. Two histories $H$ and $H'$ are *equivalent* if $txns(H) = txns(H')$ and for every transaction $T_k \in txns(H)$, $H|k = H'|k$.

We assume that shared memory locations are accessed with *read-modify-write* (rmw) primitives. A rmw primitive event on a location is *trivial* if, in any configuration, its application does not change the state of the location. Otherwise, it is called *nontrivial*.

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

#### Transaction access sets.

For a transaction $T_k$, we denote all the accounts accessed by its t-read and t-write as $Rset(T_k)$ and $Wset(T_k)$ respectively. We denote all the t-operations of a transaction $T_k$ as $Dset(T_k)$. The *read set* (resp., the *write set*) of a transaction $T_k$ in an execution $E$, denoted $Rset_E(T_k)$ (and resp. $Wset_E(T_k)$), is the set of accounts that $T_k$ attempts to read (and resp. write) by issuing a t-read (and resp. t-write) invocation in $E$ (for brevity, we sometimes omit the subscript $E$ from the notation). The *data set* of $T_k$ is $Dset(T_k) = Rset(T_k) \cup Wset(T_k)$. $T_k$ is called *read-only* if $Wset(T_k) = \emptyset$; *write-only* if $Rset(T_k) = \emptyset$ and *updating* if $Wset(T_k) \neq \emptyset$.

#### Legal reads.

Let $H$ be a t-sequential history. For every t-operation $read_k(X)$ in $H$, we define the *latest written value* of $X$ as follows: if $T_k$ contains a $write_k(X, v)$ preceding $read_k(X)$, then the latest written value of $X$ is the value of the latest such write to $X$. Otherwise, the latest written value of $X$ is the value of the argument of the latest $write_m(X, v)$ that precedes $read_k(X)$ and belongs to a committed transaction in $H$. This write is well-defined since $H$ can be assumed to start with an initial transaction ($T_{init}$) writing to all accounts.

#### Legal histories.

We say that a t-sequential history $S$ is *legal* if every t-read of an account returns the *latest written value* of this account in $S$. It means that t-sequential history $S$ is legal if all its t-reads are legal.

## 3.2 Block transactional memory

### 3.2.1 Read-write oblivious block transactional memory.

Unless otherwise specified, this paper considers BSTM implementations in which, for every execution $E$, $Dset_E(T_k)$ is unknown (to $T_k$ and every other transaction in $E$) at the start of transaction $T_k$ in $E$. More specifically, let $E$ be an execution such that $T_k \notin txns(E)$ and let $E \cdot e$ be an extension in which $T_k$ performs an invocation of associated with some account $X_i$. Then, we consider that $Dset_E(T_k)$ only contains $X_i$ in $E \cdot e$ and other accounts that might be accessed by $T_k$ in any extension of $E \cdot e$ is unknown both to $T_k$ or any other transaction in $E \cdot e$. More generally, if we consider $E$ be an execution in which $T_k$ has only invoked $m$ t-operations involving at most $m$ accounts, then we consider that $Dset_E(T_k)$ only contains these $m$ accounts in this execution.

Given a set of $n$ transactions with a *preset* order $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$, we need a deterministic concurrent execution protocol that efficiently executes the block transactions utilizing the serialization order and always leads to the same state even when executed sequentially. So, the objective is to concurrently execute the block transactions with the given preset order, but without knowing the transaction's read and write sets at start of the transaction invocation, such that concurrent execution always produces a deterministic final state that is equivalent to the state produced by sequentially executing the transactions in the preset order.

▶ **Definition 1** (Preset serializability). *Let $H$ be a history of a BSTM[1,..., n] implementation $M$. We say that $H$ is* preset serializable *if $H$ is equivalent to a legal t-sequential history $S$ that is $H_1 \cdots H_i \cdot H_{i+1} \cdots H_n$ where $H_i$ is the t-complete history of transaction $T_i$. We say that a BSTM[1,..., n] implementation $M$ is* preset serializable *if every history of $M$ is preset serializable.*

We refer to a read-write oblivious BSTM implementation with a *preset* order $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ as BSTM[1,..., n].

### 3.2.2 Read-write aware block transactional memory.

As taxonomized in Section 2, block transactional memory implementations can also provide with the read and write set of accounts right at the start of the transaction itself. This is indeed the model adopted by smart contract ecosystems like Solana [27]. For pedagogical purposes, we articulate this model although this paper is primarily concerned about the Read-write oblivious block transactional memory model.

We say that a BSTM implementation is *read-write aware* if for every execution $E$, $Rset_E(T_k)$ (and resp. $Wset_E(T_k)$) are known at the start of the transaction $T_k$ in $E$. Thus, unlike a Read-write oblivious block transactional memory, a transaction $T_k$ may be fully aware of its entire $Dset_E(T_k)$ right before invoking the first t-operation and write this to shared memory for other transactions to read.

We refer to a read-write aware BSTM implementation with a *preset* order $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$ as BSTM[($Rset(T_1)$, $Wset(T_1)$); ...; ($Rset(T_n)$, $Wset(T_n)$)].

## 4 Block-TM algorithms

This section presents algorithms for the design of read-write oblivious block transactional memory implementations.

### 4.1 A single-version algorithm

We now present the design of a preset serializable BSTM[1,..., n] implementation that maintains exactly one version of every account.

**Implementation state.** For every account $X_j$, we maintain a memory location $v_j$ that stores the value of $X_j$. Additionally for a BSTM implementation M[1,..., n], for each transaction $T_k$, we maintain a memory location Entry[$k$] which is a binary variable indicating if $T_k$ has completed.

**Read implementation.** Consider any transaction $T_k$ executed by process $p_k$. The implementation of $read_k(X_j)$ first checks if the item is already contained in $Wset(T_k)$ and if so, adopts that value. Otherwise, $T_k$ reads the value of the account from $v_j$ and adds it to its read set.

**Write implementation.** The $write_k(X, v)$ implementation simply stores (or updates if previously written to the same account) the value $v$ locally, deferring the actual update in the shared memory to $tryC_k$.

**Commit implementation.** During $tryC_k$, each transaction $T_k$ checks if Entry[$k-1$] = *true* and then if it is read-only, it simply returns $C_k$. During $tryC_k$, if $T_k$ is an updating transaction, it performs a validation of the read set and aborts the transaction if validation fails. Finally, on successful validation, optionally, if the system supports hardware transactions, it updates the write set using a hardware transaction; otherwise, $T_k$ updates its write set to the shared memory and updates Entry[$k$]= *true*. The very first transaction, $T_1$, can simply write without validation since there will be no transaction preceding it in preset, and $T_2$ cannot commit until previous $Entry[1]$ is set to *true*. Observe that, the read set invalidation only can occur because of a transaction $T_i$ preceding $T_k$ in the preset order and not one succeeding $T_k$ in the preset order.

**Algorithm 1** Strict serializable single-version BSTM[1,..., n] implementation $L$; code for transaction $T_k$

**Data: Shared locations-** $v_j$, for each account $X_j$, allows reads and writes

1 **function** $read_k(X_j)$**:**
2  **if** $X_j \notin Wset(T_k)$ **then**
   // $X_j$ is not in the $Wset_k$, read from shared memory
3  $ov_j := read(v_j)$
4  $Rset(T_k) := Rset(T_k) \cup \{X_j, ov_j\}$
5  **else**
6  $ov_j := Wset(T_k).\mathsf{locate}(X_j)$ // $X_j$ is in the $Wset(T_k)$
7  **return** $ov_j$
8 **function** $write_k(X_j, v)$**:**
9  $nv_j := v$
10  **if** $X_j \notin Wset(T_k)$ **then**
11  $Wset(T_k) := Wset(T_k) \cup \{X_j, nv_j\}$
12  **else**
   // $X_j$ is in $Wset(T_k)$, update current value to $v$
13  $Wset(T_k) := Wset(T_k).\mathsf{update}(X_j, nv_j)$
14  **return** $ok$
15 **function** $tryC_k()$**:**
16  **if** $k$=1 **then**
17  **goto** Line 22
18  **if** $Entry[k-1]$=$false$ **then**
19  **wait until** Entry$[k-1]$= $true$
20  **if** $\exists X_j \in Rset(T_k)$: $ov_j \neq read(v_j)$ **then**
21  **return** $A_k$
   // Check if system supports hardware transactions
22  **if** $\_xtest()$ **then**
23  $htm_k(Wset(T_k))$
24  **else**
25  **forall** $X_j \in Wset(T_k)$ **do**
26  $write(X_j, nv_j)$
27  $write$(Entry$[k]$,$true$)
28  **return** $C_k$
29 **function** $htm_k(Wset(T_k))$**:**
30  $ht_k = \_xbegin()$
31  **if** $ht_k.\_xbegin\_started()$ **then**
32  **forall** $X_j \in Wset(T_k)$ **do**
33  $write(X_j, nv_j)$
34  $ht_k.\_xend()$

## 4.2 Proof of correctness

Let $E$ by any finite execution of the Algorithm 1. Let $<_E$ denote a total-order on events in $E$. Let $H$ denote a subsequence of $E$ constructed by selecting *linearization points* of t-operations performed in $E$. The linearization point of a t-operation $op$, denoted as $\ell_{op}$ is associated with a memory location event or an event performed during the execution of $op$ using the following procedure.
*Completions.* First, we obtain a completion of $E$ by removing some pending invocations and adding responses to the remaining pending invocations involving a transaction $T_k$ as follows: every incomplete $read_k$, $write_k$, $tryC_k$ operation is removed from $E$ along with every aborted transaction in $E$.
*Linearization points.* We now associate linearization points to t-operations in the obtained completion of $E$ as follows: For every t-read $op_k$ that returns a non-$A_k$ value, $\ell_{op_k}$ is chosen as the event in Line 3, else, $\ell_{op_k}$ is chosen as invocation event of $op_k$. For every t-write $op_k$ that returns a non-$A_k$ value, $\ell_{op_k}$ is chosen as the event in Line 9, else, $\ell_{op_k}$ is chosen as invocation event of $op_k$. For every $op_k = tryC_k$ that returns $C_k$, $\ell_{op_k}$ is associated with Line 27. $<_H$ denotes a total-order on t-operations in the complete sequential history $H$.
*Serialization points.* The serialization of a transaction $T_j$, denoted as $\delta_{T_j}$ is associated with the linearization point of a t-operation performed during the execution of the transaction. We obtain a t-complete history $\bar{H}$ from $H$ as follows: for every transaction $T_k$ in $H$ that is complete, but not t-complete, we remove it from $H$.

$\bar{H}$ is thus a t-complete sequential history. A t-complete t-sequential history $S$ equivalent to $\bar{H}$ is obtained by associating serialization points to transactions in $\bar{H}$ as follows: If $T_k$ is an update transaction that commits, then $\delta_{T_k}$ is $\ell_{tryC_k}$. If $T_k$ is a read-only transaction in $\bar{H}$, then $\delta_{T_k}$ is assigned to the linearization point of the last t-read in $T_k$. Let $<_S$ denotes a total-order on transactions in the t-sequential history $S$.

▷ Claim 2. $S$ is legal.

**Proof.** Observe that for every $read_j(X) \to v$, there exists some transaction $T_i$ that performs $write_i(X,v)$ and completes the event in Line 9 to write $v$ as the *new value* of $X$. Note that the first transaction, $T_1$, can be committed without any read-set validation since there will be no transaction preceding it in preset order. For any updating committing transaction $T_i$, $\delta_{T_i} = \ell_{tryC_i}$. Since $read_j(X)$ returns a response $v$, the event in Line 3 must succeed the event in Line 27 when $T_i$ changes *Entry[i]* to *true*. Since $\delta_{T_i} = \ell_{tryC_i}$ precedes the event in Line 25, it follows that $\delta_{T_i} <_E \ell_{Read_j(X),}$.

We now need to prove that $\delta_{T_i} <_E \delta_{T_j}$. Consider the following cases: if $T_j$ is an updating committed transaction, then $\delta_{T_j}$ is assigned to $\ell_{tryC_j}$. But since $\ell_{read_j(X)} <_E \ell_{tryC_j}$, it follows that $\delta_{T_i} <_E \delta_{T_j}$. If $T_j$ is a read-only, then $\delta_{T_j}$ is assigned to the last t-read that returned a value. Again, it follows that $\delta_{T_i} <_E \delta_{T_j}$.

To prove that $S$ is legal, we need to show that, there does not exist any transaction $T_k$ that returns $C_k$ in $S$ and performs $write_k(X,v')$; $v' \neq v$ such that $T_i <_S T_k <_S T_j$. Now, suppose by contradiction that there exists a committed transaction $T_k$, $X \in Wset(T_k)$ that writes $v' \neq v$ to $X$ such that $T_i <_S T_k <_S T_j$.

Since $T_i$ and $T_k$ are both updating transactions that commit, $(T_i <_S T_k)$ implies that $(\delta_{T_i} <_E \delta_{T_k})$ and $(\delta_{T_i} <_E \delta_{T_k})$ implies that $(\ell_{tryC_i} <_E \ell_{tryC_k})$.

Now observe that, since $T_j$ reads the value of $X$ written by $T_i$, one of the following is true: $\ell_{tryC_i} <_E \ell_{tryC_k} <_E \ell_{read_j(X)}$ or $\ell_{tryC_i} <_E \ell_{read_j(X)} <_E \ell_{tryC_k}$.

Observe that the case that $\ell_{tryC_i} <_E \ell_{tryC_k} <_E \ell_{read_j(X)}$ is not possible. This is because the value of the account $X$ will have been overwritten by transaction $T_k$ and $read_j(X)$ should have read the value written by $T_k$ and not $T_i$—contradiction. Consequently, the only feasible case is that $\ell_{tryC_i} <_E \ell_{read_j(X)} <_E \ell_{tryC_k}$. We now need to prove that $\delta_{T_j}$ indeed precedes $\delta_{T_k} = \ell_{tryC_k}$ in $E$.

Now consider two cases:

- Suppose that $T_j$ is a read-only transaction. Then, $\delta_{T_j}$ is assigned to the last t-read performed by $T_j$. If $read_j(X)$ is not the last t-read, then there exists a $read_j(X')$ such that $\ell_{read_j(X)} <_E \ell_{tryC_k} <_E \ell_{read_j(X')}$. But then this t-read of $X'$ must abort since the value of $X$ has been updated by $T_k$ since $T_j$ first read the account $X$ (and thus would not be part of the constructed $S$)—contradiction.
- Suppose that $T_j$ is an updating transaction that commits, then $\delta_{T_j} = \ell_{tryC_j}$ which implies that $\ell_{read_j(X)} <_E \ell_{tryC_k} <_E \ell_{tryC_j}$. Then, $T_j$ must necessarily perform the validation of its read set (Line 20) and return $A_j$—contradiction.

It follows that in $S$ as constructed, every t-read returns the value of the latest written value in $S$. ◀

▷ Claim 3. If $T_i \rightarrow T_j$, then $T_i <_S T_j$.

**Proof.** This follows immediately from the assignment of serialization points. More specifically, since every transaction $T_k$ sets $Entry[k] = true$ only after transaction $T_{k-1}$ sets $Entry[k-1] = true$, it follows now, by the construction of serialization points for $S$ that the preset order is respected in $S$. ◀

▷ Claim 4. For all $i$; $1 \leq i < n$, if $T_i$ commits within a finite number of its own steps, then $T_{i+1}$ also commits within a finite number of its own steps in $E$.

**Proof.** Observe that the only spin check employed by the implementation is the checking of the *Entry* location associated with the preceding transaction in the preset order. Consequently, if $T_i$ commits by setting $Entry[i] = true$, then $T_{i+1}$ must also commit within a bounded number of steps in its execution. ◀

# 5 Single-version to multi-version

## 5.1 How multi-versioning can help?

To understand how multi-versioning will help with a BSTM implementation $M[1,\ldots,n]$, consider the scenario as shown in Figure 2, where $T_2$ is an updating transaction that performs a write of the value 1 to accounts $X_1$ and $X_2$ which starts execution after the t-read of $X_1$ by $T_1$ that returns the initial value 0 of $X_1$ following which $T_2$ performs the writes to $X_1$ and $X_2$ and commits. Observe that if we extend this execution with $T_1$ performing a t-read of $X_2$, maintaining multiple versions allows this read of $X_2$ to return the old value 0 thus preserving preset serializability, as illustrated in Figure 2b. On the other hand, a single-version implementation (Figure 2a) will necessarily need to force transaction $T_2$ to *wait* until $T_1$ finishes and additionally, perform a shared memory write to indicate the presence of the read-only transaction.

Now consider the modification of the above execution in Figure 2c, where we have $T_2$ performing a t-read of $X_1$ (returning 0) followed by the updating transaction $T_1$ running to completion and then $T_2$ resumes with a t-read of $X_2$. Note that in this scenario, it is possible for $T_2$ to validate the read set during the t-read of $X_2$, then re-try $T_2$ and return the last written values of $X_1$ and $X_2$ by updating transaction $T_1$.

## 5.2 The multi-version block transactional memory implementation

For the multi-version block transactional memory, where for each account $X_i$, we maintain a memory location $v_i$ that stores a set of tuples $([v_1,k],[v_2,k'],\ldots)$, each tuple $[v,k]$ is the value of $X_i$ and $k$ is the transaction $T_k$ that wrote that value $v$.

**Read implementation.** The implementation of $read_k(X_j)$ first checks if the account $X_j$ is already contained in $Wset(T_k)$ and if so, adopts that value. Otherwise, $T_k$ reads the value of the account $X_j$

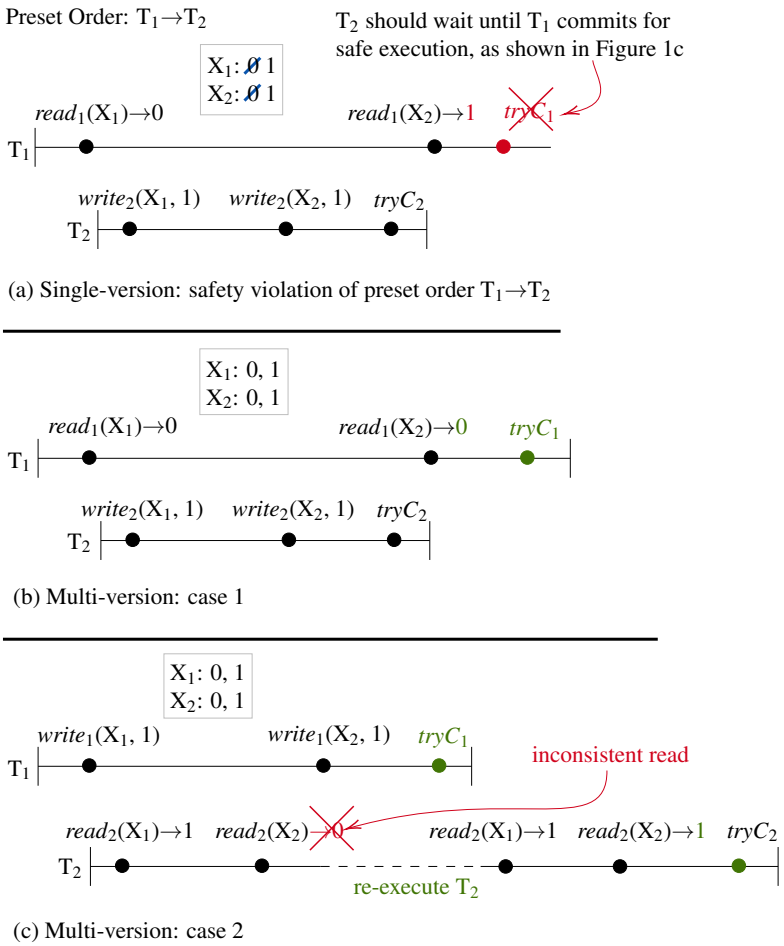


**Figure 2** Single-version to multi-version: with two transactions $T_1$ and $T_2$ and a preset order $T_1 \rightarrow T_2$, sub-figure (a) shows that when accounts $X_1$ and $X_2$ have a single version in the shared memory, the transaction $T_2$ should wait for $T_1$ to commit first for safe execution; (b) and (c) illustrate when multi-version is useful and when it is not useful while executing the transactions safely in the preset order, respectively.

from the largest version by a transaction $T_i$ such that $T_i \rightarrow T_k$ in preset order and adds it to its read set. The method *read_lvp()* finds and returns the value of the largest version created by a preceding transaction, along with the transaction identifier. The version list will never be empty since an initial version for account $X_j$ is created during initialization by $T_{init}$.

**Commit implementation.** If $T_k$ is a read-only transaction, it performs its read set validation; if any read becomes invalid by a lower-order transaction, then it returns $A_k$; otherwise, it reads the updated value and returns $C_k$. If $T_k$ is an updating transaction, it performs a validation of the read set by re-reading the corresponding versions of each account it reads and aborts the transaction if validation fails. A read set validation fails when at least one preceding transaction $T_i$ (s.t. $T_i \rightarrow T_k$) updates the new version for account $X_j$ after the version read by the $T_k$. Finally, on successful validation, optionally, if the system supports hardware transactions, it updates the write set using a hardware transaction; otherwise, $T_k$ updates its write set to the shared memory and sets the Entry[$k$]= *true*. Observe that for updating transactions, the read set invalidation can occur when a new version is created by a transaction $T_i$ preceding $T_k$ in the preset order and not one succeeding $T_k$.

### Algorithmic optimizations

We remark that the linear (in the size of the transaction's read set size) validation cost can be mitigated in some executions by employing a *global timestamp*, as employed by traditional TM implementations like TL2 [10]. The read validation is performed only if the global timestamp has changed since the start of the transaction, thus indicating the presence of a concurrent updating transaction that has committed. However, this might affect performance on Non-Uniform Memory Architectures because of lack of disjoint-access parallelism [8, 17]. This is because the timestamp will need to be updated even by transactions that do not have a read-write conflict over their respective datasets.

## 5.3 The Cost of Versioning

In this section, we derive how the preset order imposes the need to keep multiple versions of every account that is being updated by all the updating transactions. In other words, given a block of $n$ transactions, there exist executions in which $\mathcal{O}(n)$ distinct versions of every account might need to be maintained as shared memory state or read-only transactions might need to write to shared memory to inform updating transactions about their presence.

**Read invisibility** [8, 18]**.** Informally, in a TM using invisible reads, a transaction cannot reveal any information about its read set to other transactions. Thus, given an execution $E$ and some transaction $T_k$ with a non-empty read set, transactions other than $T_k$ cannot distinguish $E$ from an execution in which $T_k$'s read set is empty. This prevents TMs from applying nontrivial primitives during t-read operations and from announcing read sets of transactions during *tryC*.

Formally, a TM implementation $M$ uses *invisible reads* if for every execution $E$ of $M$: for every read-only transaction $T_k \in txns(E)$, no event of $E|k$ is nontrivial in $E$.

▶ **Definition 5.** *Let $E$ be any execution of a TM implementation $M$. We say that $E$* maintains *$c$* values $\{v_1, \ldots, v_c\}$ of account $X$, *if there exists an execution $E \cdot E'$ of $M$ such that*

- *$E'$ contains the complete executions of $c$ t-reads of account $X$ and,*
- *for all $i \in \{1, \ldots, c\}$, the response of the $i^{th}$ t-read of $X$ in $E'$ is $v_i$, and if the response of the $i^{th}$ t-read of $X$ in $E'$ is $r \neq v_i$, then $E \cdot E'$ is not an execution of $M$.*

We say that a Block-STM implementation $M$ provides *sequential TM-progress* if for every finite execution $E$ of $M$ in which every transaction is t-complete, every transaction $T_k$ that runs t-sequentially is committed.

▶ **Theorem 6.** *Consider any BSTM[1,..., n] implementation with sequential TM-progress. Then $M$ has an execution in which it uses $n-1$ versions of each account accessed by some read-only transaction or the read-only transaction does not have read-invisibility.*

**Proof.** As illustrated in Figure 3, consider an execution $E$ that consists of a read-only transaction $T_1$ that performs a t-read of account $X_1$ that returns the initial value 0 of $X_1$. We will assume that the read-only transaction $T_1$ is invisible. Immediately following the t-read, consider the following extension:

- The t-complete execution of an updating transaction $T_2$ that writes value 1 to account $X_1$ and writes value 1 to account $X_2$. Since $T_1$ is invisible (by assumption), transaction $T_2$ must commit (by assumption of liveness).
- Now we extend this execution with a transaction $T_3$ that performs a t-read of $X_1$ (this must return the value 1).
- Following this, we introduce a new updating transaction $T_4$ that writes value 2 to accounts $X_1$ and $X_2$.

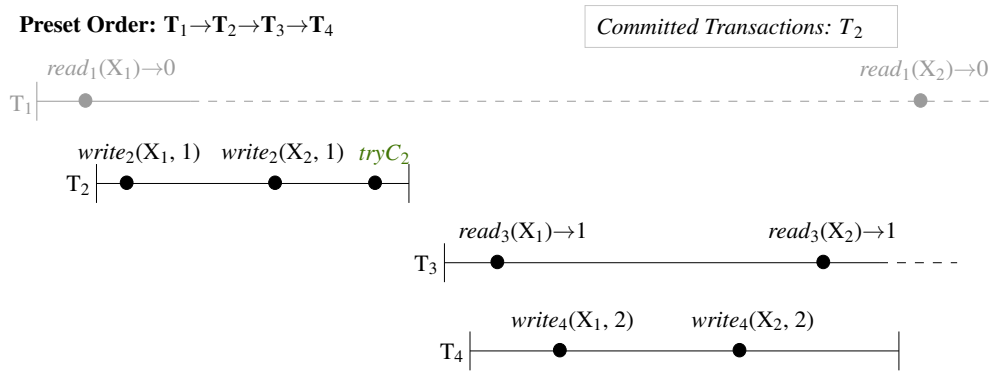


**Figure 3** Invisible read and safe execution: example showing four transactions $T_1$, $T_2$, $T_3$, $T_4$ and a preset order $T_1 \to T_2 \to T_3 \to T_4$, where $T_1$ and $T_3$ are read-only transactions, while $T_2$ and $T_4$ are updating transactions. $T_1$ is an invisible read transaction, while $T_2$ is a committed transaction. The execution is showing the values read and written by transactions during safe execution in preset order.

- Now we extend read-only transaction $T_3$ that performs a t-read of $X_2$. Since $T_3$ precedes $T_4$ in the preset order, we require that the value returned by the t-read of $X_2$ be the latest written value of $X_2$ in the t-sequential history $T_1 \cdot T_2 \cdot T_3$. Consequently, the only value that the t-read of $X_2$ by $T_3$ can return in any extension of this execution is the value 1 (observe that this is true even if $T_3$ aborts and re-runs in a step-contention free execution).
- Immediately following the t-complete execution of $T_4$, transaction $T_1$ resumes execution and performs a t-read of $X_2$.

Since $T_1$ is the first transaction in the preset order, the only value this t-read can return is the initial value 0. Observe that, in this execution, we require that we need both versions of $X_2$ (0 and 1) to be maintained. ◀

### Implication for Read-write aware block transactional memory

We now argue that the lower bound on versioning applies also to a restricted class of read-write aware BSTM implementations. We consider the set of *strict data-partitioned* implementations [16] which intuitively forces two transactions which access disjoint sets of accounts to not access a common memory location. Intuitively, this restriction when applied to a BSTM[($Rset(T_1)$, $Wset(T_1)$); ...; ($Rset(T_n)$, $Wset(T_n)$)] makes this only as useful as the read-write oblivious model.

Formally, a BSTM implementation $M$ is *strict data-partitioned* [16], if for every account $X$, there exists a set of memory locations $Loc_M(X)$ such that

- for any two accounts $X_1, X_2$; $Loc_M(X_1) \cap Loc_M(X_2) = \emptyset$,
- for every execution $E$ of $M$ and every transaction $T \in txns(E)$, every memory location accessed by $T$ in $E$ is contained in $Loc_M(X)$ for some $X \in Dset(T)$
- for all executions $E$ and $E'$ of $M$, if $E|X = E'|X$ for some account $X$, then the configurations after $E$ and $E'$ only differ in the states of the memory locations in $Loc_M(X)$. Here, $E|X$ denote the subsequence of the execution $E$ derived by removing all events associated with account $X$.

To understand how the construction in Theorem 6 can be applied to strict data-partitioned block transactional memory implementations, consider the execution involving two transactions $T_1$ and $T_2$: $T_1$ reads account $X_1$, then $T_2$ performs writes of new values to accounts $X_1$ and $X_2$. Observe that, by assumption of invisible reads, such an execution exists. By assumption of strict data-partitioning, the t-read of $X_1$ by $T_1$ cannot access memory locations associated with $X_2$ and thus, cannot *pre-read* the values associated with $X_2$. Consequently, when we extend this execution with the t-read of account $X_2$ by $T_1$, the initial value of $X_2$ (prior to the update by $T_2$) must be returned. Thus, intuitively, both

versions of $X_2$ must be maintained. Extending this argument to the construction of the multi-phased execution in Theorem 6, the proof follows.

**Remarks.** We remark that, while the definition of strict data-partitioning might seem overly restrictive (for e.g., global timestamps or process-specific shared memory locations might not be used), we conjecture that it is possible to extend the equivalence to a more practical model of block transactional memory implementations (analogous to those developed for traditional in-memory transactions [1, 7] that could incorporate the delineation of "value" of the account from the "metadata" associated with shared memory synchronization).

## 6 Discussion

*Reduced hardware transactions* Intel shared memory architectures support hardware transactions that allow the *atomic* execution of a set of transactional reads and writes. Specifically, current CPUs have included instructions to mark a block of memory accesses as transactional, allowing them to be executed *atomically* in hardware. Hardware transactions typically provide automatic conflict detection at cacheline granularity, thus ensuring that a transaction will be aborted if it experiences memory contention on a cacheline. Typically, if the hardware transaction is executed sequentially, i.e., in the absence of step-contention, the transaction will commit.

In the read-write oblivious block transactional memory model, the ordering curse that comes from the preset order prevents a transaction $T_j$ from committing before transaction $T_i$ when $T_i \rightarrow T_j$ (as explained in Figure 1). Thus, the execution of a set of transactional writes may almost need to be carried out sequentially, thus perfectly supporting the use of hardware transactions. As described in Algorithm 1 (Lines 22, 23, 30-34), the writes of the values in a transaction's write set are carried by a hardware transaction and should succeed with high probability. It is an open question if hardware transactions can be exploited in a more comprehensive way for block transactional memory, akin to attempts with traditional TMs [10, 16, 26].

*Relaxing preset serializability* The definition of preset serializability (Definition 1) requires reads of accounts to return the latest written value to that account respecting the preset order. However, certain workloads in smart contract ecosystems might not require such restrictions allowing for the possibility of relaxed account semantics. This might allow the development of different algorithmic techniques and yield block transactional memory implementations with better performance.

*Equivalence between read-write aware and read-write oblivious models* For any parallel execution approach in both the *read-write aware* and *read-write oblivious* settings, identifying the conflicts between transactions and resolving them efficiently is the key to increased transaction throughput [6, 13, 14, 15, 19, 30]. As we have articulated in Section 2, smart contract parallel execution models seek to provide hints about transactions to the parallel execution engine so enable potentially faster execution, thus overcoming the restrictions of respecting the preset order. Both Solana [27] and Sui [29] equip the transactions with *a priori* state access information about the read-write sets along with potential transactional conflicts (in the case of the latter). Whether these models yield concrete complexity separation results against the read-write oblivious model remains an outstanding open question.

Read-write aware models may require more effort from developers to avoid the aborts of optimistic execution of read-write oblivious settings. However, it allows for more localized dynamic *gas* fee marketplaces; if dependencies are specified a priori, transactions occurring in a congested contract of the blockchain state might be processed separately from others; to prevent a localized state hotspot from increasing fees for the whole blockchain network. For example, a popular *non fungible token (NFT)* mint could create a large number of transaction requests in a short period of time [20]. A

blockchain based on a read-write aware setting can detect state hotspots upfront, such as the NFT minting, to rate limit and charge a higher fee for transactions containing them [28, 14]. This enables ordinary transactions to execute promptly, while transactions related to the minting process are prioritized independently based on the total gas associated with them and the resulting congestion.

## References


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

2 M. J. Amiri, D. Agrawal, and A. El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1337–1347, Los Alamitos, CA, USA, jul 2019. IEEE, IEEE Computer Society.

3 P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 83–92. IEEE, IEEE Computer Society, Feb 2019.

4 P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani. Optsmart: a space efficient optimistic concurrent execution of smart contracts. *Distributed and Parallel Databases*, pages 1–53, 2022.

5 The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. `https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf`, August 2022. [August 11, 2022].

6 Arcology: A blockchain ecosystem with unlimited scalability. `https://doc.arcology.network/arcology-concurrency-control`, 2021. [Online: accessed 24 January 2024].

7 H. Attiya and E. Hillel. The cost of privatization in software transactional memory. *IEEE Trans. Computers*, 62(12):2531–2543, 2013.

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

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

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

11 T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312, New York, NY, USA, 2017. PODC '17, ACM.

12 Ethereum (ETH): open-source blockchain-based distributed computing platform. `https://www.ethereum.org/`. [Online: accessed 15 January 2024].

13 M. Fouda. The case for parallel processing chains. `https://medium.com/alliancedao/the-case-for-parallel-processing-chains-90bac38a6ba4`, September 2022. [Online: accessed 10 January 2024].

14 S. Foundation. All about parallelization. `https://blog.sui.io/parallelization-explained/`, January 2024. [Online: accessed 23 January 2024].

15 R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery.

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

17 A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, pages 151–160, New York, NY, USA, 1994. Association for Computing Machinery.


18 P. Kuznetsov and S. Ravi. On partial wait-freedom in transactional memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, page 10, New York, NY, USA, 2015. Association for Computing Machinery.

19 M. Labs. Parallel execution & monad. https://medium.com/monad-labs/parallel-execution-monad-f4c203cddf31. [Online: accessed 10 January 2024].

20 What is lazy minting? introducing a smarter yet economical way to mint nfts. https://www.antiersolutions.com/what-is-lazy-minting-introducing-a-smarter-yet-economical-way-to-mint-nfts/, January 2024. [Online: accessed 13 January 2024].

21 D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, New York, NY, USA, 2010. Association for Computing Machinery.

22 M. Piduguralla, S. Chakraborty, P. S. Anjana, and S. Peri. Dag-based efficient parallel scheduler for blockchains: Hyperledger sawtooth as a case study. In *European Conference on Parallel Processing*, pages 184–198, Berlin, Heidelberg, 2023. Springer, Springer-Verlag.

23 V. Saraph and M. Herlihy. An empirical study of speculative concurrency in ethereum smart contracts. In *International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019)*, pages 4:1–4:15, Dagstuhl, Germany, 2019. OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

24 Architecture documentaion v1.2. https://sawtooth.hyperledger.org/docs/1.2/architecture/transaction_scheduling.html. [Online: accessed 10 January 2024].

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

26 N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, page 204–213, New York, NY, USA, 1995. Association for Computing Machinery.

27 Solana documentation. https://docs.solana.com/. [Online: accessed 10 January 2024].

28 Solana fees, part 1. https://www.umbraresearch.xyz/writings/solana-fees-part-1, December 2023. [Online: accessed 10 December 2023].

29 Sui documentation: Discover the power of sui through examples, guides, and concepts. https://docs.sui.io. [Online: accessed 10 January 2024].

30 Umbraresearch. Lifecycle of a solana transaction. https://www.umbraresearch.xyz/writings/lifecycle-of-a-solana-transaction. [Online: accessed 15 January 2024].

31 A. Yakovenko. Sealevel - parallel processing thousands of smart contracts. https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, September 2019. [Online: accessed 23 January 2024].