

Block-STM*

Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing

Rati Gelashvili
Aptos

Alexander Spiegelman
Aptos

Zhuolun Xiang
Aptos

George Danezis
Mysten Labs & UCL

Zekun Li
Aptos

Dahlia Malkhi
Chainlink Labs

Yu Xia
MIT

Runtian Zhou
Aptos

Abstract

Block-STM is a parallel execution engine for smart contracts, built around the principles of Software Transactional Memory. Transactions are grouped in blocks, and every execution of the block must yield the same deterministic outcome. Block-STM further enforces that the outcome is consistent with executing transactions according to a preset order, leveraging this order to dynamically detect dependencies and avoid conflicts during speculative transaction execution. At the core of Block-STM is a novel, low-overhead collaborative scheduler of execution and validation tasks.

Block-STM is implemented on the main branch of the Diem Blockchain code-base and runs in production at Aptos. Our evaluation demonstrates that Block-STM is adaptive to workloads with different conflict rates and utilizes the inherent parallelism therein. Block-STM achieves up to 110k tps in the Diem benchmarks and up to 170k tps in the Aptos Benchmarks, which is a 20x and 17x improvement over the sequential baseline with 32 threads, respectively. The throughput on a contended workload is up to 50k tps and 80k tps in Diem and Aptos benchmarks, respectively.

1 Introduction

A central challenge facing emerging decentralized web3 platforms and applications is improving the throughput of the underlying Blockchain systems. At the core of a Blockchain system is state machine replication, allowing a set of entities to agree on and apply a sequence of *blocks* of transactions. Each transaction contains *smart contract* code to be executed, and every entity that executes the block of transactions must arrive at the same final state. While there has been progress on scaling parts of the system, Blockchains are still bottlenecked by other components, such as transaction execution.

Our goal is to accelerate the in-memory execution of transactions via parallelism. Transactions that access different memory locations can always be executed in parallel. However, in a Blockchain system transactions can have significant number of access conflicts. This may happen due to potential

performance attacks, accessing popular contracts or due to economic opportunities (such as auctions and arbitrage [16]).

Conflicts are the main challenge for performance. An approach pioneered by Software Transactional Memory (STM) libraries [29, 44] is to instrument memory accesses to detect conflicts. STM libraries with optimistic concurrency control [17] (OCC) record memory accesses, *validate* every transaction post execution, and abort and re-execute transactions when validation surfaces a conflict. The final outcome is equivalent to executing transactions sequentially in some order. This equivalent order is called *serialization*.

Prior works [5, 8, 18] have capitalized on the specifics of the Blockchain use-case to improve on the STM performance. Their approach is to pre-compute dependencies in a form of a directed acyclic graph of transactions that can be executed via a fork-join schedule. The resulting schedule is dependency-aware, and avoids corresponding conflicts. If entities are incentivized to record and share the dependency graph, then some entities may be able to avoid the pre-computation overhead.

In the context of deterministic databases, BOHM [21] demonstrated a way to avoid pre-computing the dependency graph. BOHM assumes that the write-sets of all transactions are known prior to execution, and enforces a specific preset serialization of transactions. As a result, each read is associated with the last write preceding it in that order. Using a multi-version data-structure [10], BOHM executes transactions when their read dependencies are resolved, avoiding corresponding conflicts.

Our contribution. We present Block-STM, an in-memory smart contract parallel execution engine built around the principles of optimistically controlled STM. Block-STM does not require a priori knowledge of transaction write-sets, avoids pre-computation, and accelerates transaction execution autonomously without requiring further communication. Similar to BOHM, Block-STM uses multi-version shared data-structure and enforces a preset serialization. The final outcome is equivalent to the sequential execution of transactions in the preset order in which they appear in the block.

*Rati Gelashvili, Alexander Spiegelman, and Zhuolun Xiang share first authorship. Contact emails: gelash@aptoslabs.com, sasha.spiegelman@gmail.com, xiangzhuolun@gmail.com. The work was initiated while all authors were working at Novi at Meta.

The key observation is that with OCC and a preset serialization, when a transaction aborts, its write-set can be used to efficiently detect future dependencies. This has two advantages with respect to pre-execution: (1) in the optimistic case when there are few conflicts, most transactions are executed once, (2) otherwise, write-sets are likely to be more accurate as they are based on a more up-to-date execution. Another advantage of the of the preset order is that it allows as comprehensive correctness testing as we can compare to a sequential execution output.

Two observations that contribute to the performance of Block-STM in the Blockchain context are the following. First, in blockchain systems, the state is updated per block. This allows the Block-STM to avoid the synchronization cost of committing transactions individually. Instead Block-STM lazily commits all transactions in a block based on two atomic counters and a double-collect technique [9]. Second, transactions are specified in smart contract languages, such as Move [11] and Solidity [51], and run in a virtual machine that encapsulates their execution and ensures safe behavior. Therefore, opacity [24] is not required, allowing Block-STM to efficiently combine an optimistic concurrent control with multi-version data structure, without additional mechanisms to avoid reaching inconsistent states.

The main challenge in combining OCC and preset serialization is that validations are no longer independent from each other and must logically occur in a sequence. A failed validation of a transaction implies that all higher transactions can be committed only if they get successfully validated afterwards. Block-STM handles this issue via a novel collaborative scheduler that optimistically dispatches execution and validation tasks, prioritizing tasks for transactions lower in the preset order. While concurrent priority queues are notoriously hard to scale across threads [4, 40], Block-STM capitalizes on the preset serialization order and the boundedness of transaction indices to implement a concurrent ordered set abstraction using only a few shared atomic counters.

We provide comprehensive correctness proofs for both Safety and Liveness, proving that no deadlock or livelock is possible and the final state is always equivalent to the state produced by executing the transactions sequentially.

A Rust implementation of Block-STM is merged on the main branches of the Diem [47] and its successor Aptos [2] open source blockchain code-bases [1, 3]. The experimental evaluation demonstrates that Block-STM outperforms sequential execution by up to 20x on low-contention workloads and by up to 9x on high-contention ones. Importantly, Block-STM suffers from at most 30% overhead when the workload is completely sequential. In addition, Block-STM significantly outperforms a state-of-the-art deterministic STM [52] implementation, and performs closely to BOHM which requires perfect write-sets information prior to execution.

The rest of the paper is organized as following: Section 2 provides a high-level overview of Block-STM. Section 3 describes the full algorithm, while Section 4 describes Block-STM implementation and evaluation. Section 5 discusses related work and Section 6 concludes the paper. Appendix A contains the comprehensive correctness proofs.

2 Overview

The input of Block-STM is a block of transactions, denoted by BLOCK, containing n transactions, which defines the preset serialization order $tx_1 < tx_2 < \dots < tx_n$. The problem definition is to execute the block and produce the final state equivalent to the state produced by executing the transactions in sequence tx_1, tx_2, \dots, tx_n , each tx_j executed to completion before tx_{j+1} is started. The goal is to utilize available threads to produce such final state as efficiently as possible.

Each transaction in Block-STM might be executed several times and we refer to the i^{th} execution as *incarnation i* of a transaction. We say that an incarnation is *aborted* when the system decides that a subsequent re-execution with an incremented incarnation number is needed. A *version* is a pair of a transaction index and an *incarnation number*. To support reads and writes by transactions that may execute concurrently, Block-STM maintains an in-memory multi-version data structure that separately stores for each memory location the latest value written per transaction, along with the associated transaction version. When transaction tx reads a memory location, it obtains from the multi-version data-structure the value written to this location by the highest transaction that appears before tx in the preset serialization order, along with the associated version. For example, transaction tx_5 can read a value written by transaction tx_3 even if transaction tx_6 has written to same location. If no smaller transaction has written to a location, then the read (e.g. all reads by tx_1) is resolved from storage based on the state before the block execution.

For each incarnation, Block-STM maintains a *write-set* and a *read-set*. The read-set contains the memory locations that are read during the incarnation, and the corresponding versions. The write-set describes the updates made by the incarnation as (memory location, value) pairs. The write-set of the incarnation is applied to shared memory (the multi-version data-structure) at the end of execution. After an incarnation executes it needs to pass validation. The validation re-reads the read-set and compares the observed versions. Intuitively, a successful validation implies that writes applied by the incarnation are still up-to-date, while a failed validation implies that the incarnation has to be aborted.

Dependency estimation. Block-STM does not pre-compute dependencies. Instead, for each transaction, Block-STM treats the write-set of an aborted incarnation as an estimation of the write-set of the next one. Together with the multi-version data structure and the preset order it allows reducing the

Check done: if V and E are empty and no other thread is performing a task, then return.

Find next task: Perform the task with the smallest transaction index tx in V and E :

1. **Execution task:** Execute the next incarnation of tx . If a value marked as ESTIMATE is read, abort execution and add tx back to E . Otherwise:
 - (a) If there is a write to a memory location to which the previous finished incarnation of tx has not written, create validation tasks for all transactions $\geq tx$ that are not currently in E or being executed and add them to V .
 - (b) Otherwise, create a validation task only for tx and add it to V .
2. **Validation task:** Validate the last incarnation of tx . If validation succeeds, continue. Otherwise, **abort**:
 - (a) Mark every value (in the multi-versioned data-structure) written by the incarnation (that failed validation) as an ESTIMATE.
 - (b) Create validation tasks for all transactions $> tx$ that are not currently in E or being executed and add them to V .
 - (c) Create an execution task for transaction tx with an incremented incarnation number, and add it to E .

Figure 1. High level scheduling

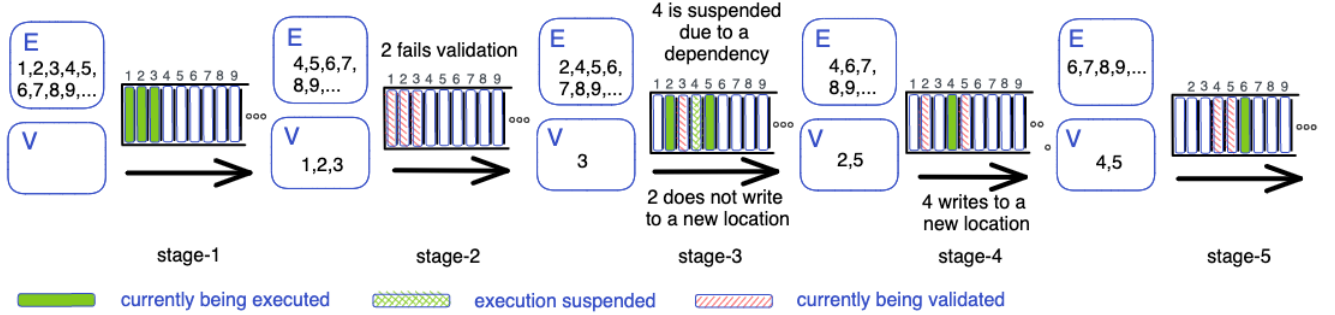


Illustration of an example execution of the abstract Block-STM collaborative scheduler.

Initially, all transactions are in the ordered set E . In this example, transaction tx_4 depends on tx_2 . In stage 1, since there are no validation tasks, the threads execute transactions tx_1, tx_2, tx_3 in parallel. Then, in stage 2, the threads validate transactions tx_1, tx_2, tx_3 in parallel, the validation of tx_2 fails and the validations of tx_1 and tx_3 succeed. The incarnation of tx_2 is aborted, each of its writes is marked as an ESTIMATE in the multi-version data-structure, the next incarnation task is added to E , and a new validation task for tx_3 is added to V . In stage 3, transaction tx_3 is validated and transactions tx_2 and tx_4 start executing their respective incarnations. However, the execution of tx_4 reads a value marked as ESTIMATE, is aborted due to the dependency on tx_2 and the thread executes the next transaction in E , which is tx_5 . As explained above, tx_4 is recorded as a dependency of tx_2 and added back to E when tx_2 's incarnation finishes. After both tx_2 and tx_5 finish execution, the corresponding validation tasks are added to V . In this example, the incarnation of tx_2 does not write to a memory location to which its previous incarnation did not write. Therefore, another validation of tx_3 is not required. In stage 4, tx_2 and tx_5 are successfully validated and tx_4 is executed. From this point on, tx_1, tx_2 , and tx_3 will never be re-executed as there is no task associated with them in V or E (and no task associated with a higher transaction may lead to creating it). The execution of tx_4 writes to a new memory location, and thus tx_5 is added to V for re-validation. In stage 5, transactions tx_4 and tx_5 are validated and transaction tx_6 is executed.

abort rate by efficiently detecting potential dependencies. When an incarnation is aborted due to a validation failure, the entries in the multi-version data-structure corresponding to its write-set are replaced with a special ESTIMATE marker. This signifies that the next incarnation is estimated to write to the same memory location. In particular, an incarnation of transaction tx_j stops and is immediately aborted whenever it reads a value marked as an ESTIMATE that was written by a lower transaction tx_k . This is an optimization to abort an incarnation early when it is likely to be aborted in the future due to a validation failure, which would happen if the next incarnation of tx_k would indeed write to the same location (the ESTIMATE markers that are not overwritten are removed by the next incarnation).

Collaborative scheduler. Block-STM introduces a collaborative scheduler, which coordinates the validation and execution tasks among threads. The preset serialization order dictates that the transactions must be committed in order, so a successful validation of an incarnation does not guarantee

that it can be committed. This is because an abort and re-execution of an earlier transaction in the block might invalidate the incarnation read-set and necessitate re-execution. Thus, when a transaction aborts, all higher transactions are scheduled for re-validation. The same incarnation may be validated multiple times, by different threads, and potentially in parallel, but Block-STM ensures that only the first abort per version succeeds (the rest are ignored).

Since transactions must be committed in order, the Block-STM scheduler prioritizes tasks (validation and execution) associated with lower-indexed transactions. Next, we overview the high-level ideas behind the approach. The detailed logic is described in Section 3 and formally proved in Appendix A.

Abstractly, the Block-STM collaborative scheduler tracks an ordered set V of pending validation tasks and an ordered set E of pending execution tasks. Initially, V is empty and E contains execution tasks for the initial incarnation of all transactions in the block. A transaction $tx \notin E$ is either currently being executed or (its last incarnation) has completed. On a high level, each thread repeats the instructions described in Figure 1.

When a transaction tx_k reads an ESTIMATE marker written by tx_j (with $j < k$), we say that tx_k encounters a *dependency*. We treat tx_k as tx_j 's dependency because its read depends on a value that tx_j is estimated to write. For the ease of presentation, in the above description a transaction is added back to E immediately upon encountering a dependency. However, as explained in Section 3, Block-STM implements a slightly more involved mechanism. Transaction tx_k is first recorded separately as a dependency of tx_j , and only added back to E when the next incarnation of tx_j completes (i.e. when the dependency is resolved).

The ordered sets, V and E , are each implemented via a single atomic counter coupled with a mechanism to track the status of transactions, i.e. whether a given transaction is ready for validation or execution, respectively. To pick a task, threads increment the smaller of these counters until they find a task that is ready to be performed. To add a (validation or execution) task for transaction tx , the thread updates the status and reduces the corresponding counter to tx (if it had a larger value). For presentation purposes, the above description omits an optimization that the Block-STM scheduler uses in cases 1(b) and 2(c), where instead of reducing the counter value, the new task is returned.

Optimistic validation. An incarnation of transaction might write to a memory location that was previously read by an incarnation of a higher transaction according to the preset serialization order. This is why in 1(a), when an incarnation finishes, new validation tasks are created for higher transactions. Importantly, validation tasks are scheduled optimistically, e.g. it is possible to concurrently validate the latest incarnations of transactions tx_j , tx_{j+1} , tx_{j+2} and tx_{j+4} . Suppose transactions tx_j , tx_{j+1} and tx_{j+4} are successfully validated, while the validation of tx_{j+2} fails. When threads are available, Block-STM capitalizes by performing these validations in parallel, allowing it to detect the validation failure of tx_{j+2} faster in the above example (at the expense of a validation of tx_{j+4} that needs to be redone). Identifying validation failures and aborting incarnations as soon as possible is crucial for the system performance, as any incarnation that reads values written by a incarnation that aborts also needs to be aborted, forming a cascade of aborts.

When an incarnation writes only to a subset of memory locations written by the previously completed incarnation of the same transaction, i.e. case 1(b), Block-STM schedules validation just for the incarnation. This is sufficient due to 2(a), as the whole write-set of the previous incarnation is marked as estimates during the abort. The abort leads to optimistically creating validation tasks for higher transactions in 2(b). Threads that perform these tasks can already detect validation failures due to the ESTIMATE markers on memory locations, instead of waiting for a subsequent incarnation to finish.

Commit rule. In [23], we derive a precise predicate for when transaction tx_j can be considered committed (its roughly

when an incarnation is successfully validated after lower transactions $0, \dots, j - 1$ have already been committed). It would be possible to continuously track this predicate, but to reduce the amount of work and synchronization involved, the Block-STM scheduler only checks whether the entire block of transactions can be committed. This is done by observing that there are no more tasks to perform and at the same time, no threads that are performing any tasks.

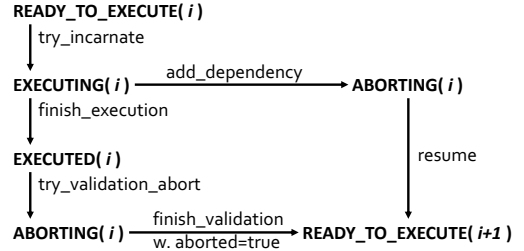


Figure 2. Illustration of status transitions

3 Block-STM Detailed Description

In this section, we describe Block-STM. Upon spawning, threads perform the `run()` procedure in Line 1. Our pseudo-code is divided into several modules that the threads use. The **Scheduler** module contains the shared variables and logic used to dispatch execution and validation tasks. The **MVMemory** module contains shared memory in a form of a multi-version data-structure for values written and read by different transactions in Block-STM. Finally, the **VM** module describes how reads and writes are instrumented during transaction execution.

Block-STM finishes when all threads join after returning from the `run()` invocation. At this point, the output of Block-STM can be obtained by calling the `MVMemory.snapshot()` function that returns the final values for all affected memory locations. This function can be easily parallelized and the output can be persisted to main storage (abstracted as a **Storage** module), but these aspects are out of the scope here.

3.1 High-Level Thread Logic

We start by the high-level logic described in Algorithm 1. The `run()` procedure interfaces with the **Scheduler** module and consists of a loop that lets the invoking thread continuously perform available validation and execution tasks. The thread looks for a new task in Line 9, and dispatches a proper handler based on its kind, i.e. function `try_execute` in Line 5 for an `EXECUTION_TASK` and function `needs_reexecution` in Line 7 for a `VALIDATION_TASK` (since, as discussed in Section 2, a successful validation does not change state, while failed validation implies that the transaction requires re-execution). Both of this functions take a transaction version (transaction index and incarnation number) as an input. A `try_execute` function invocation may return a new validation task back to the caller, and a `needs_reexecution` function invocation may return a new execution task.

Algorithm 1 Thread logic

```
1: procedure run()
2:    $task \leftarrow \perp$ 
3:   while  $\neg$ Scheduler.done() do
4:     if  $task \neq \perp \wedge task.kind = EXECUTION\_TASK$  then
5:        $task \leftarrow try\_execute(task.version)$  ▷ returns a validation task, or  $\perp$ 
6:     if  $task \neq \perp \wedge task.kind = VALIDATION\_TASK$  then
7:        $task \leftarrow needs\_reexecution(task.version)$  ▷ returns a re-execution task, or  $\perp$ 
8:     if  $task = \perp$  then
9:        $task \leftarrow Scheduler.next\_task()$ 
10: function try_execute(version) ▷ returns a validation task, or  $\perp$ 
11:    $(txn\_idx, incarnation\_number) \leftarrow version$ 
12:    $vm\_result \leftarrow VM.execute(txn\_idx)$  ▷ VM does not write to shared memory
13:   if  $vm\_result.status = READ\_ERROR$  then
14:     if  $\neg$ Scheduler.add_dependency( $txn\_idx, vm\_result.blocking\_txn\_idx$ ) then
15:       return try_execute(version) ▷ dependency resolved in the meantime, re-execute
16:     return  $\perp$ 
17:   else
18:      $wrote\_new\_location \leftarrow MVMemory.record(version, vm\_result.read\_set, vm\_result.write\_set)$ 
19:     return Scheduler.finish_execution( $txn\_idx, incarnation\_number, wrote\_new\_location$ )
20: function needs_reexecution(version) ▷ returns a task for re-execution, or  $\perp$ 
21:    $(txn\_idx, incarnation\_number) \leftarrow version$ 
22:    $read\_set\_valid \leftarrow MVMemory.validate\_read\_set(txn\_idx)$ 
23:    $aborted \leftarrow \neg read\_set\_valid \wedge Scheduler.try\_validation\_abort(txn\_idx, incarnation\_number)$ 
24:   if  $aborted$  then
25:      $MVMemory.convert\_writes\_to\_estimates(txn\_idx)$ 
26:   return Scheduler.finish_validation( $txn\_idx, aborted$ )
```

3.1.1 Execution Tasks. An execution task is processed using the `try_execute` procedure. First, a `VM.execute` function is invoked in Line 12. As discussed in Section 3.2.1, by the `VM` design, this function reads from memory (`MVMemory` data-structure and the main `Storage`), but never modifies any state while being performed. Instead, a successful `VM` execution returns a *write-set*, consisting of memory locations and their updated values, which are applied to `MVMemory` by the `record` function invocation in Line 18. In Block-STM, `VM.execute` also captures and returns a *read-set*, containing all memory locations read during the incarnation, each associated with whether a value was read from `MVMemory` or `Storage`, and in the former case, the version of the transaction execution that previously wrote the value. The read-set is also passed to the `MVMemory.record` call in Line 18 and stored in `MVMemory` for later validation purposes.

Every `MVMemory.record` invocation returns an indicator whether a write occurred to a memory location not written to by the previous incarnation of the same transaction. As discussed in Section 2, in Block-STM this indicator determines whether the higher transactions (than the transaction that just finished execution, in the preset serialization order) require further validation. `Scheduler.finish_execution` in Line 19 schedules the required validation tasks. When a new location is not written, `wrote_new_location` variable is set to *false* and it suffices to only validate the transaction itself.

In this case, due to an internal performance optimization, the `Scheduler` module sometimes returns this validation task back to the caller from the `finish_execution` invocation.

The `VM` execution of transaction tx_j may observe a read dependency on a lower transaction tx_k in the preset order, $k < j$. As discussed in Section 2, this happens when the last incarnation of tx_k wrote to a memory location that tx_j reads, but when the incarnation of tx_k aborted before the read by tx_j . In this case, the index k of the blocking transaction is returned as `vm_result.blocking_txn_idx`, a part of the output in Line 12. In order to re-schedule the execution task for tx_j for after the blocking transaction tx_k finishes its next incarnation, `Scheduler.add_dependency` is called in Line 14. This function returns *false* if it encounters a race condition when tx_k gets re-executed before the dependency can be added. The execution task is then retried immediately in Line 15.

3.1.2 Validation Tasks. A `validate_read_set` call in Line 22 obtains the last read-set recorded by an execution of txn_idx and checks that re-reading each memory location in the read-set still yields the same values. To be more precise, for every value that was read, the read-set stores a read descriptor. This descriptor contains the version of the transaction (during the execution of which the value was written), or \perp if the value was read from storage (i.e. not written by a

Algorithm 2 The **MVMemory** module

Atomic Variables:

```
data ← Map, initially empty           ▷ (location, txn_idx) maps to a pair (incarnation_number, value), or to an ESTIMATE marker.
last_written_locations ← Array(BLOCK.size(), {}) ▷ txn_idx to a set of memory locations written during its last finished execution.
last_read_set ← Array(BLOCK.size(), {})   ▷ txn_idx to a set of (location, version) pairs per reads in last finished execution.
27: procedure apply_write_set(txn_index, incarnation_number, write_set)
28:   for every (location, value) ∈ write_set do
29:     data[(location, txn_idx)] ← (incarnation_number, value)           ▷ store in the multi-version data structure
30: function rcu_update_written_locations(txn_index, new_locations)
31:   prev_locations ← last_written_locations[txn_idx]                   ▷ loaded atomically (RCU read)
32:   for every unwritten_location ∈ prev_locations \ new_locations do
33:     data.remove((unwritten_location, txn_idx))                       ▷ remove entries that were not overwritten
34:   last_written_locations[txn_idx] ← new_locations                   ▷ store newly written locations atomically (RCU update)
35:   return new_locations \ prev_locations ≠ {}                       ▷ was there a write to a location not written the last time
36: function record(version, read_set, write_set)
37:   (txn_idx, incarnation_number) ← version
38:   apply_write_set(txn_idx, incarnation_number, write_set)
39:   new_locations ← {location | (location, ★) ∈ write_set}           ▷ extract locations that were newly written
40:   wrote_new_location ← rcu_update_written_locations(txn_idx, new_locations)
41:   last_read_set[txn_idx] ← read_set                                 ▷ store the read-set atomically (RCU update)
42:   return wrote_new_location
43: procedure convert_writes_to_estimates(txn_idx)
44:   prev_locations ← last_written_locations[txn_idx]                 ▷ loaded atomically (RCU read)
45:   for every location ∈ prev_location do
46:     data[(location, txn_idx)] ← ESTIMATE                             ▷ entry is guaranteed to exist
47: function read(location, txn_idx)
48:   S ← {(location, idx), entry} ∈ data | idx < txn_idx}
49:   if S = {} then
50:     return (status ← NOT_FOUND)
51:   ((location, idx), entry) ← arg max_idx S
52:   if entry = ESTIMATE then
53:     return (status ← READ_ERROR, blocking_txn_idx ← idx)
54:   return (status ← OK, version ← (idx, entry.incarnation_number),
value ← entry.value)
55: function snapshot()
56:   ret ← {}
57:   for every location | ((location, ★), ★) ∈ data do
58:     result ← read(location, BLOCK.size())
59:     if result.status = OK then
60:       ret ← ret ∪ {location, result.value}
61:   return ret
62: function validate_read_set(txn_idx)
63:   prior_reads ← last_read_set[txn_idx]
64:   for every (location, version) ∈ prior_reads do
65:     cur_read ← read(location, txn_idx)
66:     if cur_read.status = READ_ERROR then
67:       return false
68:     if cur_read.status = NOT_FOUND ∧ version ≠ ⊥ then
69:       return false
70:     if cur_read.status = OK ∧ cur_read.version ≠ version then
71:       return false
72:   return true
```

smaller transaction). The incarnation numbers are monotonically increasing, so it is sufficient to validate the read-set by comparing the corresponding descriptors.

If validation fails, `try_validation_abort` on `Scheduler` is called in Line 23, which returns an indicator of whether the abort was successful. `Scheduler` ensures that only one failing validation per version may lead to a successful abort.

Hence, if `abort_validation` returns `false`, then the incarnation was already aborted. If the abort was successful, then `convert_writes_to_estimates(txn_idx)` in Line 25 is called, which replaces the write-set of the aborted version in the shared memory data-structure with special ESTIMATE markers. A successful abort leads to scheduling the transaction for re-execution and the higher transactions for validation during the `Scheduler.finish_validation` call in Line 26.

Algorithm 3 The **VM** module

```
73: function execute(txn_id)
74:   read_set  $\leftarrow$  {} ▷ (location, version) pairs
75:   write_set  $\leftarrow$  {} ▷ (location, value) pairs
76:   run transaction BLOCK[txn_idx] ▷ run transaction, intercept reads and writes
77:   ....
78:   upon writing value at a memory location:
79:     if (location, prev_value)  $\in$  write_set then ▷ store the latest value per location
80:       write_set  $\leftarrow$  write_set \ {(location, prev_value)}
81:       write_set  $\leftarrow$  write_set  $\cup$  {(location, value)} ▷ VM does not write to MVMemory or Storage
82:     ....
83:   upon reading a memory location:
84:     if (location, value)  $\in$  write_set then ▷ value written by this txn
85:       VM reads value
86:     else
87:       result  $\leftarrow$  MVMemory.read(location, txn_idx)
88:       if result.status = NOT_FOUND then
89:         read_set  $\leftarrow$  read_set  $\cup$  {(location,  $\perp$ )} ▷ record version  $\perp$  when reading from storage
90:         VM reads from Storage
91:       else if result.status = OK then
92:         read_set  $\leftarrow$  read_set  $\cup$  {(location, result.version)}
93:         VM reads result.value
94:       else
95:         return result ▷ return (READ_ERROR, blocking_txn_id) from the VM.execute
96:     ....
97:   return (read_set, write_set)
```

Sometimes, (as an optimization), the re-execution task is returned (that proceeds to return the new version from `needs_reexecution` and then in Line 5 become the only thread to execute the next incarnation of the transaction).

3.2 Multi-Version Memory

The **MVMemory** module (Algorithm 2) describes the shared memory data-structure in Block-STM. It is called *multi-version* because it stores multiple writes for each memory location, along with a value and an associated version of a corresponding transaction. In the pseudo-code, we represent the main data-structure, called *data*, with an abstract map interface, mapping (*location*, *txn_idx*) pairs to the corresponding entries, which are (*incarnation_number*, *value*) pairs. In order to support a read of memory *location* by transaction *tx_j*, *data* provides an interface that returns an entry written at *location* by the transaction with the highest index *i* such that *i* < *j*. This functionality is used in Line 48 and Line 51. For clarity of presentation, our pseudo-code focuses on the abstract functionality of the map, while standard concurrent data-structure design techniques can be used for an efficient implementation (discussed in Section 4).

For every transaction, **MVMemory** stores a set of memory locations in the *last_written_locations* array and a set of (*location*, *version*) pairs in the *last_read_set* array. We assume that these sets are loaded and stored atomically, which can

be accomplished by storing a pointer to the set and accessing the pointer atomically, i.e. via the read-copy-update [33].

Recording. The record function takes a transaction version along with the read-set and the write-set (resulting from the execution of the version). The write-set consists of (memory location, value) pairs that are applied to the *data* map by `apply_write_set` procedure invocation. The invocation of `rcu_update_written_locations` that follows in Line 40 updates *last_written_locations* and also removes (in Line 33) from the *data* map all entries at memory locations that were not overwritten by the latest write-set of the transaction (i.e. locations in the *last_written_locations* before, but not after the update). This function also determines and returns whether a new memory location was written (i.e. in *last_written_locations* after, but not before the update). This indicator is stored in *wrote_new_location* variable and returned from the record function. Before returning, the read-set of the transaction is stored in the *last_read_set* array via an RCU pointer update.

The `convert_writes_to_estimates` procedure, called during a transaction abort, iterates over *last_written_locations* of the transaction, and replaces each stored (*incarnation_number*, *value*) pair with a special ESTIMATE marker. It ensures that validations fail for higher transactions if they have read the data written by the aborted incarnation. While removing the entries can also accomplish this, the ESTIMATE marker also serves as a “write estimate” for the next incarnation of this

Algorithm 4 The Scheduler module, variables, utility APIs and next task logic

Atomic variables:
 $execution_idx \leftarrow 0, validation_idx \leftarrow 0, decrease_cnt \leftarrow 0, num_active_tasks \leftarrow 0, done_marker \leftarrow false$

▷ Respectively: An index that tracks the next transaction to try and execute. A similar index for tracking validation. Number of times

$validation_idx$ or $execution_idx$ was decreased. Number of ongoing validation and execution tasks. Marker for completion.

 $txn_dependency \leftarrow \text{Array}(\text{BLOCK.size}(), \text{mutex}(\{\}))$

▷ txn_idx to a mutex-protected set of dependent transaction indices

 $txn_status \leftarrow \text{Array}(\text{BLOCK.size}(), \text{mutex}((0, \text{READY_TO_EXECUTE})))$

▷ txn_idx to a mutex-protected pair ($incarnation_number, status$),

where $status \in \{\text{READY_TO_EXECUTE}, \text{EXECUTING}, \text{EXECUTED}, \text{ABORTING}\}$.

```

98: procedure decrease_execution_idx(target_idx)           123:   idx_to_execute ← execution_idx.fetch_and_increment()
99:   execution_idx ← min(execution_idx, target_idx)   ▷ atomic 124:   return try_incarnate(idx_to_execute)
100:   decrease_cnt.increment()                          125: function next_version_to_validate()
101: function done()                                     126:   if validation_idx ≥ BLOCK.size() then
102:   return done_marker                                127:     check_done()
103: procedure decrease_validation_idx(target_idx)       128:     return ⊥
104:   validation_idx ← min(validation_idx, target_idx) ▷ atomic 129:   num_active_tasks.increment()
105:   decrease_cnt.increment()                          130:   idx_to_validate ← validation_idx.fetch_and_increment()
106: procedure check_done()                              131:   if idx_to_validate < BLOCK.size() then
107:   observed_cnt ← decrease_cnt                       132:     (incarnation_number, status) ← txn_status[idx_to_validate].lock()
108:   if min(execution_idx, validation_idx) ≥ BLOCK.size() ∧    133:     if status = EXECUTED then
   num_active_tasks = 0 ∧ observed_cnt = decrease_cnt then 134:       return (idx_to_validate, incarnation_number)
109:   done_marker ← true                                135:   num_active_tasks.decrement()
110: function try_incarnate(txn_idx)                      136:   return ⊥
111:   if txn_idx < BLOCK.size() then                   137: function next_task()
112:     with txn_status[txn_idx].lock()                 138:   if validation_idx < execution_idx then
113:       if txn_status[txn_idx].status = READY_TO_EXECUTE then 139:     version_to_validate ← next_version_to_validate()
114:         txn_status[txn_idx].status ← EXECUTING          140:     if version_to_validate ≠ ⊥ then
115:         return (txn_idx, txn_status[txn_idx].incarnation_number) 141:       return (version ← version_to_validate,
116:   num_active_tasks.decrement()                        kind ← VALIDATION_TASK)
117:   return ⊥                                           142:   else
118: function next_version_to_execute()                   143:     version_to_execute ← next_version_to_execute()
119:   if execution_idx ≥ BLOCK.size() then              144:     if version_to_execute ≠ ⊥ then
120:     check_done()                                       145:       return (version ← version_to_execute,
121:     return ⊥                                           kind ← EXECUTION_TASK)
122:   num_active_tasks.increment()                        146:   return ⊥

```

transaction. Any transaction that observes an ESTIMATE of transaction tx when reading during a speculative execution, waits for the dependency to resolve (tx to be re-executed), as opposed to ignoring the ESTIMATE and likely aborting if tx 's next incarnation again writes to the same memory location.

Reads. The `MVMemory.read` function takes a memory location and a transaction index txn_idx as its input parameters. First, it looks for the highest transaction index, idx , among transactions lower than txn_idx that have written to this memory location (Line 48 and Line 51). Based on the fixed serialization order of transactions in the block, this is the best guess for reading speculatively (writes by transactions lower than idx are overwritten by idx , and the speculative premise is that the transactions between idx and txn_idx do not write to the same memory location). The value written by transaction idx is returned in Line 54, alongside with

the full version (i.e. idx and the incarnation number) and an OK status. However, if the entry corresponding to transaction idx is an ESTIMATE marker, then the read returns an `READ_ERROR` status and idx as a blocking transaction index. This is an indication for the caller to postpone the execution of transaction txn_idx until the next incarnation of the blocking transaction idx completes. Essentially, at this point, it is estimated that transaction idx will perform a write that is relevant for the correct execution of transaction txn_idx .

When no lower transaction has written to the memory location, a read returns a `NOT_FOUND` status, implying that the value cannot be obtained from the previous transactions in the block. As we will describe shortly, the caller can then complete the speculative read by reading from storage.

Algorithm 5 The Scheduler module, dependencies and finish logic

```
147: function add_dependency(txn_idx, blocking_txn_idx)
148:   with txn_dependency[blocking_txn_idx].lock()
149:     if txn_status[blocking_txn_idx].lock().status = EXECUTED then                                     ▶ thread holds 2 locks
150:       return false                                                                                               ▶ dependency resolved before locking in Line 148
151:     txn_status[txn_idx].lock().status ← ABORTING                                                           ▶ previous status must be EXECUTING
152:     txn_dependency[blocking_txn_idx].insert(txn_idx)
153:     num_active_tasks.decrement()                                                                           ▶ execution task aborted due to a dependency
154:   return true
155: procedure set_ready_status(txn_idx)
156:   with txn_status[txn_idx].lock()
157:     (incarnation_number, status) ← txn_status[txn_idx]                                                     ▶ status must be ABORTING
158:     txn_status[txn_idx] ← (incarnation_number + 1, READY_TO_EXECUTE)
159: procedure resume_dependencies(dependent_txn_indices)
160:   for each dep_txn_idx ∈ dependent_txn_indices do
161:     set_ready_status(dep_txn_idx)
162:   min_dependency_idx ← min(dependent_txn_indices)                                                         ▶ minimum is ⊥ if no elements
163:   if min_dependency_idx ≠ ⊥ then
164:     decrease_execution_idx(min_dependency_idx)                                                           ▶ ensure dependent indices get re-executed
165: procedure finish_execution(txn_idx, incarnation_number, wrote_new_path)
166:   txn_status[txn_idx].lock().status ← EXECUTED                                                             ▶ status must have been EXECUTING
167:   deps ← txn_dependency[txn_idx].lock().swap({})                                                         ▶ swap out the set of dependent transaction indices
168:   resume_dependencies(deps)
169:   if validation_idx > txn_idx then                                                                       ▶ otherwise index already small enough
170:     if wrote_new_path then
171:       decrease_validation_idx(txn_idx)                                                                     ▶ schedule validation for txn_idx and higher txns
172:     else
173:       return (version ← (txn_idx, incarnation_number), kind ← VALIDATION_TASK)
174:   num_active_tasks.decrement()
175:   return ⊥                                                                                                   ▶ no task returned to the caller
176: function try_validation_abort(txn_idx, incarnation_number)
177:   with txn_status[txn_idx].lock()
178:     if txn_status[txn_idx] = (incarnation_number, EXECUTED) then
179:       txn_status[txn_idx].status ← ABORTING                                                               ▶ thread changes status, starts aborting
180:       return true
181:   return false
182: procedure finish_validation(txn_idx, aborted)
183:   if aborted then
184:     set_ready_status(txn_idx)
185:     decrease_validation_idx(txn_idx + 1)                                                                     ▶ schedule validation for higher transactions
186:     if execution_idx > txn_idx then                                                                       ▶ otherwise index already small enough
187:       new_version ← try_incarnate(txn_idx)
188:       if new_version ≠ ⊥ then
189:         return (new_version, kind ← EXECUTION_TASK)                                                     ▶ return re-execution task to the caller
190:       num_active_tasks.decrement()                                                                           ▶ done with validation task
191:   return ⊥                                                                                                   ▶ no task returned to the caller
```

The `validate_read_set` function loads (via RCU) the most recently recorded read-set from the transaction's execution in Line 63. The function calls `read` for each location and checks observed status and version against the read-set (recall that version \perp in the read-set means that the corresponding prior read returned `NOT_FOUND` status, i.e. it read a value from [Storage](#)). As we saw in Section 3.1.2, `validate_read_set` function is invoked during validation

in Line 22, at which point the incarnation that is being validated is already executed and has recorded the read-set. However, if the thread performing a validation task for incarnation i of a transaction is slow, it is possible that `validate_read_set` function invocation observes a read-set recorded by a later (i.e. $> i$) incarnation. In this case, incarnation i is guaranteed to be already aborted (else higher incarnations would never start), and the validation task will

have no effect on the system regardless of the outcome (only validations that successfully abort affect the state and each incarnation can be aborted at most once).

The snapshot function is called after Block-STM finishes, and returns the value written by the highest transaction for every location that was written to by some transaction.

3.2.1 VM execution. In Algorithm 3 we describe how reads and writes are handled in Block-STM by the `VM.execute` function (invoked while performing an execution task, in Line 12). This function tracks and returns the transaction’s read- and write-sets, both initialized to empty. When a transaction attempts to write a value to a location, the (location, value) pair is added to the write-set, possibly replacing a pair with a prior value (if it is not the first time the transaction wrote to this location during the execution).

When a transaction attempts to read a location, if the location is already in the write-set then the `VM` reads the corresponding value (that the transaction itself wrote) in Line 85. Otherwise, `MVMemory.read` is performed. If it returns `NOT_FOUND`, then `VM` reads the value directly from storage (abstracted as a `Storage` module that contains values preceding the block execution) and records $(location, \perp)$ in the read-set. If `MVMemory.read` returns `READ_ERROR`, then `VM` execution stops and returns the error and the blocking transaction index (for the dependency) to the caller. If it returns `OK`, then `VM` reads the resulting value from `MVMemory` and records the location and version pair in the read-set.

Note that for simplicity of presentation, if the transaction reads the same location more than once, the pseudo-code repeats the read and makes separate record in the read-set. Even if reading the same location results in reading different values, Block-STM algorithm maintains correctness because all reads are eventually validated and the `VM` captures the errors that may arise due to any opacity violations.

3.3 Scheduling

The `Scheduler` module contains the necessary state and synchronization logic for managing the execution and validation tasks. For each transaction in a block, the `txn_status` array contains the most up-to-date incarnation number (initially 0) and the status of this incarnation, which can be one of `READY_TO_EXECUTE` (initial value), `EXECUTING`, `EXECUTED` and `ABORTING`. The entries of the `txn_status` array are protected by a lock to provide atomicity.

Status transitions are illustrated in Figure 2. The thread that changes the status from `READY_TO_EXECUTE` to `EXECUTING` in Line 114 when incarnation number is i performs incarnation i of the transaction. The status never becomes `READY_TO_EXECUTE(i)` again, guaranteeing that no incarnation is performed more than once. Afterwards, this thread sets the status to `EXECUTED(i)` in Line 166. Similarly, only the thread that changes the status from `EXECUTED(i)` to `ABORTING(i)` returns `true` from `try_validation_abort` for incarnation i . After performing the steps associated with a

successful abort, as discussed in Section 3.1.2, this thread then updates the status to `READY_TO_EXECUTE(i + 1)` in Line 158. This indicates that an execution task for incarnation $i + 1$ is ready to be created.

When incarnation i of transaction tx_k aborts because of a read dependency on transaction tx_j ($j < k$ in the preset serialization order), the status of tx_k is updated to `ABORTING(i)` in Line 151. The corresponding `add_dependency(k, j)` invocation returns `true` and Block-STM guarantees that some thread will subsequently finish executing transaction tx_j and resolve tx_k ’s dependency in Line 158 (called from Line 161) by setting its status to `READY_TO_EXECUTE(i + 1)`.

The `txn_dependency` array is used to track transaction dependencies. In the above example, when transaction tx_k reads an estimate of transaction tx_j and calls `add_dependency(k, j)` (that returns `true`), k is added to `txn_dependency[j]` in Line 152. Our pseudo-code explicitly describes lock-based synchronization for the dependencies stored in the `txn_dependency` array. This is to demonstrate the handling of a race between the `add_dependency` function of tx_k and the `finish_execution` procedure of tx_j (in particular, to guarantee that transaction tx_j will always clear its dependencies in Line 167). The problematic scenario could arise if after tx_k observed the read dependency, transaction tx_j raced to `finish_execution` and cleared its dependencies. However, due to the check in Line 149, dependency will not be added and the `add_dependency` invocation will return `false`. Then, the status of tx_k would remain `EXECUTING` and the caller would immediately re-attempt the execution task of tx_k , incarnation i , in Line 15.

Management Tasks. Block-STM scheduler maintains `execution_idx` and `validation_idx` atomic counters. Together, one can view the status array and the validation (or execution) index counter as a counting-based implementation of an ordered set abstraction for selecting lowest-indexed available validation (or execution) task.

The `validation_idx` counter tracks the index of the next transaction to be validated. A thread picks an index in Line 130 in the `next_version_to_validate` function by performing the `fetch_and_increment` instruction on the `validation_idx`. It then checks if the transaction with the corresponding index is ready to be validated (i.e. the status is `EXECUTED`), and if it is, determines the latest incarnation number. A similar `execution_idx` counter is used in combination with the status array to manage execution tasks. In the `next_version_to_execute` function, a thread picks an index by `fetch_and_incrementing` in Line 123, then invokes the `try_incarnate` function. Only if the transaction is in a `READY_TO_EXECUTE` state, this function will set the status to `EXECUTING` and return the corresponding version for execution.

When transaction status is updated to `READY_TO_EXECUTE`, Block-STM ensures that the corresponding execution task eventually gets created. In the `resume_dependencies` procedure, the execution index is reduced by the call in Line 164

to be no higher than indices of all transactions that had a dependency resolved. In `finish_validation` function after a successful abort, however, there may be a single re-execution task (unless the task was already claimed by another thread after the status was set, something that is checked in Line 188). As an optimization, instead of reducing `execution_idx`, the execution task is sometimes returned to the caller in Line 189.

Similarly, if a validation of transaction tx_k was successfully aborted, then Block-STM ensures, in the `finish_validation` function (in Line 185), that $validation_idx \leq k$. In addition, in the `finish_execution` function of transaction tx_k , Block-STM invokes `decrease_validation_idx` in Line 171 if a new memory location was written by the associated incarnation. Otherwise, only a validation task for tx_k is created that may be returned to the caller.

Finally, the `next_task` function decides whether to obtain a version to execute or version to validate based on a simple heuristic, by comparing the two indices in Line 138.

Detecting Completion. The `Scheduler` provides a mechanism for the threads to detect when all execution and validation tasks are completed. This is not trivial because individual threads might obtain no available tasks from the `next_task` function, but more execution and validation tasks could still be created later, e.g. if a validation task that is being performed by another thread fails.

Block-STM implements a `check_done` procedure that determines when all work is completed and the threads can safely return. In this case, a `done_marker` is set to `true`, providing a cheap way for all threads to exit their main loops in Line 3. Threads invoke a `check_done` procedure in Line 120 and Line 127, when observing an execution or validation index that is already $\geq \text{BLOCK.size}()$. In the following, we explain the logic behind `check_done`.

A straw man approach would be to check that both execution and validation indices are at least as large as the `BLOCK.size()`. The first problem with this approach is that it does not consider when the execution and validation tasks actually finish. For example, the `validation_idx` may be incremented in Line 130 and become `BLOCK.size()`, but it would be incorrect for the threads to return, as the corresponding validation task of transaction `BLOCK.size() - 1` may still fail. To overcome this problem, Block-STM utilizes the `num_active_tasks` atomic counter to track the number of ongoing execution and validation tasks. Then, in addition to the indices, the scheduler also checks whether `num_active_tasks = 0` in Line 108.

The `num_active_tasks` counter is incremented in Line 122 and Line 129, right before `execution_idx` and `validation_idx` are *fetch-and-increment*-ed, respectively. The `num_active_tasks` is decremented if no task corresponding to the fetched index is created (Line 116 and Line 135), or after the tasks finish (Line 174 and Line 190). As an optimization, when `finish_execution` or `finish_validation` functions return

a new task to the caller, `num_active_tasks` is left unchanged (instead of incrementing and decrementing that cancel out).

The second challenge is that `validation_idx`, `execution_idx` and `num_active_tasks` are separate counters, e.g. it is possible to read that `validation_idx` has value `BLOCK.size()`, then read that `num_active_tasks` has value 0, without these variables simultaneously holding the respective values. Block-STM handles this by another counter, `decrease_cnt`, incremented in `decrease_execution_idx` and `decrease_validation_idx` procedures (Line 100, Line 105). By reading `decrease_cnt` twice in `check_done`, it is possible to detect if validation or execution index decreases from their observed values when `num_active_tasks` is read to be 0.

4 Implementation and Evaluation

Our Block-STM implementation is in Rust, and is merged on the main branch of the open source Diem and Aptos projects [2, 47]. Both Blockchains run a virtual machine for smart contracts in Move language [11]. The `VM` captures all execution errors that could stem from inconsistent reads during speculative transaction execution. The `VM` also caches the reads from `Storage`. Importantly, the preset order allows us to test correctness by comparing to sequential implementation outputs.

Diem VM does not support suspending transaction execution at the exact point when a read dependency is encountered. Instead, when a transaction is aborted due to a `READ_ERROR`, it is later (after the dependency is resolved) restarted from scratch. Aptos VM supports this feature.

To mitigate the impact of restarting `VM` execution from scratch, we check the read-set of the previous incarnation for dependencies before the `VM.execute` invocation in Line 12.

Another related optimization implemented in Block-STM occurs when the `Scheduler.add_dependency` invocation returns `false` in Line 14. This indicates that the dependency has been resolved. Instead of Line 15 (that would restart the execution from scratch with the Diem VM), Block-STM calls `add_dependency` from the `VM` itself, and can thus re-read and continue execution when `false` is returned.

Block-STM implementation uses the standard cache padding technique to mitigate false sharing. The logic for `num_active_tasks` is implemented using the Resource Acquisition Is Initialization (RAII) design pattern. Finally, Block-STM implements the `data` map in `MVMemory` as a concurrent hashmap over access paths, with lock-protected search trees for efficient `txn_idx`-based look-ups.

4.1 Experimental Results

We evaluated Block-STM on a Amazon Web Services c5a.16xlarge instance (AMD EPYC CPU and 128GB memory) with Ubuntu 18.04 operating system. The experiments run on a single socket with up to 32 physical cores without hyper-threading.

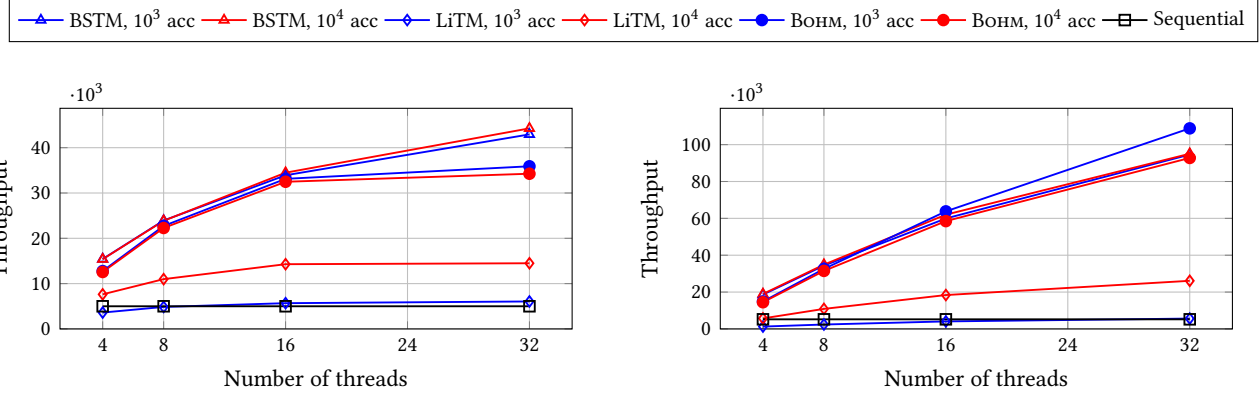


Figure 3. Comparison of BSTM, LiTM, BOHM and sequential execution for block size 10^3 (left) and 10^4 (right). BOHM is provided with perfect write estimates. Diem p2p txns.

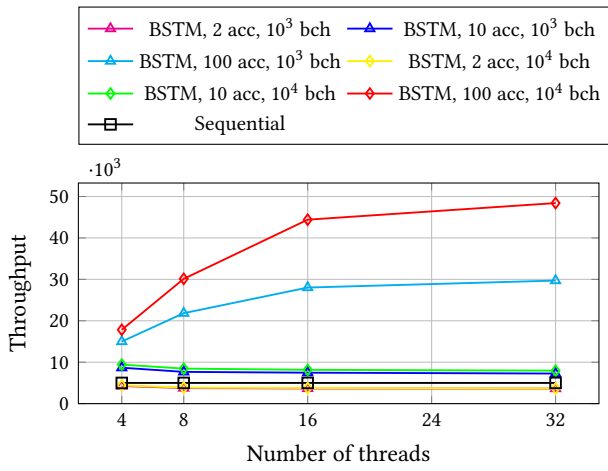


Figure 4. Comparison of BSTM and sequential execution for block size 10^3 and 10^4 , account sizes 2, 10 and 100. Diem p2p transactions.

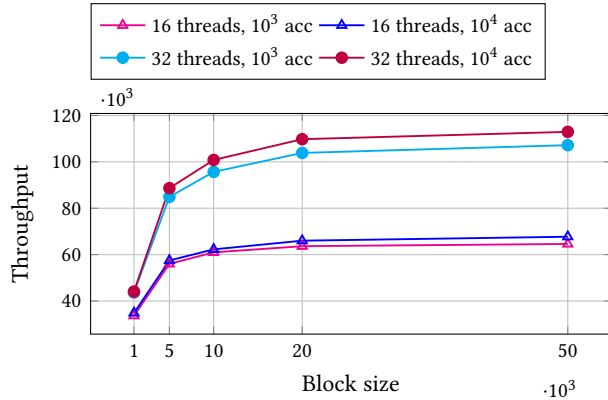


Figure 5. Throughput of BSTM for various block sizes. Diem p2p transactions.

The evaluation benchmark executes the whole block, consisting of peer-to-peer (p2p) transactions implemented in Move. Each p2p transaction randomly chooses two different accounts and performs a payment.

We first perform experiments with *Diem* p2p transactions¹ that perform 21 reads and 4 writes. For a Diem p2p transaction from account *A* to account *B*, the 4 writes of the transaction involve updating balances and sequence numbers of *A* and *B*. The reason for 21 reads is that every Diem transaction is verified against some on-chain information to decide whether the transaction should be processed, some of which is specific to p2p transactions. During this process, information such as the correct block time and whether or not the account is frozen is read.

We also perform experiments with *Aptos* p2p transactions² that perform 8 reads and 5 writes each, where the *Aptos* p2p transactions reduce many of the verification and on-chain reads mentioned above. The VM execution overhead of a single Diem p2p compared to a single *Aptos* p2p is about 100%, as will be shown in Figure 3 and Figure 6, the throughput of sequentially executing Diem and *Aptos* p2p transaction is about 5k and 10k, respectively. We experiment with block sizes of 10^3 and 10^4 transactions and the number of accounts of 2, 10, 100, 10^3 and 10^4 . The number of accounts determines the amount of conflicts, and in particular, with just 2 accounts the load is inherently sequential (each transaction depends on the previous one). Each data point is an average of 10 measurements.

This reported measurements include the cost of reading all required values from storage, and computing the outputs (i.e. all affected paths and the final values), but not persisting the outputs to *Storage*. The outputs are computed according to the *MVMemory.snapshot* logic, but parallelized (per affected memory locations).

We compare Block-STM to BOHM [21] and LiTM [52]. BOHM is a deterministic database engine that enforces a preset order by assuming transactions' write-sets are known. BOHM has a pre-execution phase in which it uses the write-sets information to build a multi-version data-structure that

¹ <https://github.com/danielxiangzl/Block-STM>

² <https://github.com/danielxiangzl/Block-STM/tree/aptos>

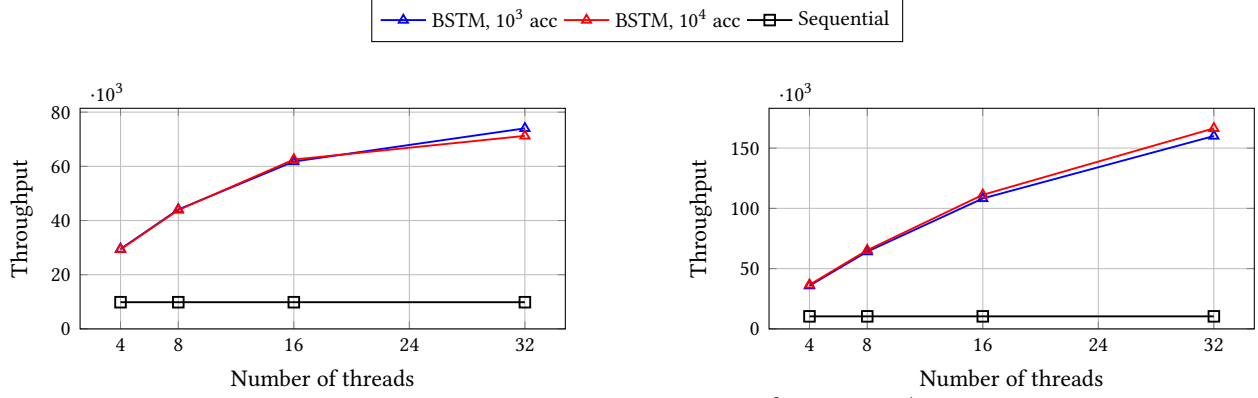


Figure 6. Comparison of BSTM and Sequential execution for block size 10^3 (left) and 10^4 (right). Aptos p2p transactions.

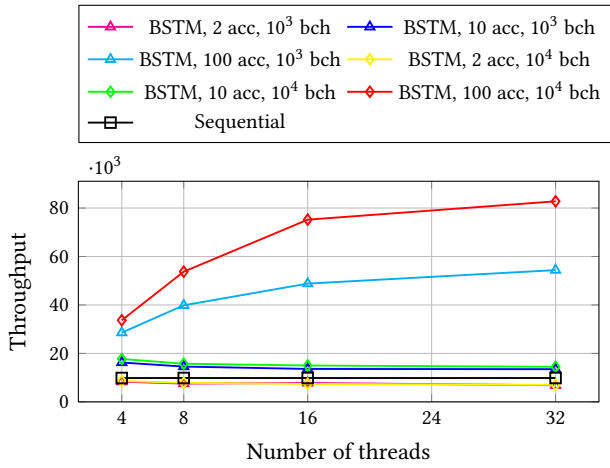


Figure 7. Comparison of BSTM and sequential execution for block size 10^3 and 10^4 , account sizes 2, 10 and 100. Aptos p2p transactions.

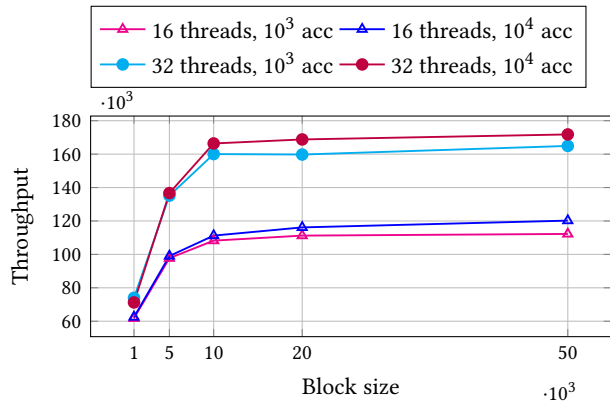


Figure 8. Throughput of BSTM for various block sizes. Aptos p2p transactions.

captures the dependencies with respect to the preset order. Then, BOHM executes transactions in parallel, delays any transaction that has unresolved read dependencies by buffering it in a concurrent queue, and resumes the execution once the dependencies are resolved. Note that in the

Blockchain use-case the assumption of knowing all write-sets in advance is not realistic, so to compare Block-STM to BOHM we artificially provide BOHM with perfect write-sets information. Note that our measurements of BOHM only include parallel execution but not the write-sets analysis, thus would be significantly better than the performance of BOHM in practice when the write-sets analysis time is non-negligible. LiTM [52], a recent deterministic STM library, claims to outperform other deterministic STM approaches on the Problem Based Benchmark Suite [45]. We describe LiTM in more detail in Section 5. In order to have a uniform setting for comparison, we implemented both a variant of BOHM³ and LiTM⁴ in Rust in the Diem Blockchain.

The Block-STM comparison to BOHM, LiTM and sequential baseline for Diem p2p transactions is shown in Figure 3. The Block-STM comparison to sequential baseline for Aptos p2p transactions is shown in Figure 6. We will open source all our implementations and benchmarks to enable reproducible results.

Comparison to BOHM [21]. The results show that Block-STM has comparable throughput to BOHM in most cases, and is significantly better with 32 threads and 10^3 block size. Since BOHM relies on perfect write-sets information and thus perfect dependencies among all transactions, it can delay the execution of a transaction after all its dependencies have been executed, avoiding the overhead of aborting and re-execution. In contrast, Block-STM require no information about write dependencies prior to execution and therefore will incur aborts and re-execution. Still, the performance of Block-STM is comparable to BOHM, implying the abort rates of Block-STM is substantially small, thanks to the run-time write-sets estimation and the low-overhead collaborative scheduler. We also found the overhead of constructing the multi-version data-structure of BOHM significant compared to Block-STM, without which BOHM’s throughput will be slightly better than Block-STM.

³<https://github.com/danielxiangzl/Block-STM/tree/bohm>

⁴<https://github.com/danielxiangzl/Block-STM/tree/litm>

Comparison to LiTM [52]. With 10^4 accounts, Block-STM has around 3-4x speedup over LiTM regardless of the block size or transactions type (standard or simplified). With 10^3 accounts, the speedup is larger (up to 25x) over LiTM, confirming that Block-STM is less sensitive to conflicts.

Comparison to sequential execution. For Diem and Aptos benchmarks, Block-STM scales almost perfectly under low contention, achieving up to 90k tps and 160k tps, which is 18x and 16x over the sequential execution, respectively.

Comparison under highly contended workload. Figure 4 and Figure 7 reports Block-STM evaluation results with highly contended workloads. With a completely sequential workload (2 accounts) Block-STM has at most 30% overhead vs the sequential execution in both Diem and Aptos benchmarks. With 10 accounts Block-STM already outperforms the sequential execution and with 100 accounts Block-STM gets up to 8x speedup in both benchmarks. Note that with 100 accounts Block-STM does not scale beyond 16 threads, suggesting that 16 threads already utilize the inherent parallelism in such a highly contended workload.

Maximum throughput of Block-STM We also evaluate Block-STM with increasing block sizes (up to 50k) to find the maximum throughput of Block-STM in Figure 5 and Figure 8. For 32 threads, Block-STM achieves up to 110k tps for Diem p2p (21x speedup over sequential) and 170k tps for Aptos p2p (17x speedup over sequential). For 16 threads, Block-STM achieves up to 67k tps for Diem p2p (13x speedup) and 120k tps for Aptos p2p (12x speedup).

Conclusion. Our evaluation demonstrates that Block-STM is adaptive to workload contention and utilizes the inherent parallelism therein. For Aptos benchmark, it achieves over 160k tps on workloads with low contention, over 80k on workloads with high contention, and at most 30% overhead on workload that are completely sequential.

5 Related Work

The STM approach. The problem of atomically executing transactions in parallel in shared memory has been extensively studied in the literature in the past few decades in the context of STM libraries (e.g., [17, 19, 22, 25, 28, 29, 44]). These libraries instrument the concurrent memory accesses associated with different transactions, detect and deal with conflicts, and provide the final outcome equivalent to executing transactions sequentially in some serialization order. In the STM libraries based on optimistic concurrency control [17, 31], threads repeatedly speculatively execute and validate transactions. A successful validation commits and determines the transaction position in the serialization order.

By default, STM libraries do not guarantee the same outcome when transactions are re-executed multiple times. This is unsuitable for Blockchain systems, as validators need to agree on the outcome of block execution. Deterministic STM libraries [36, 38, 49] guarantee a unique final state.

Due to required conflict bookkeeping and aborts, general-purpose STM libraries often suffer from performance limitations compared to custom-tailed solutions and are rarely deployed in production [14]. However, STM performance can be dramatically improved by restricting it to specific use-cases [20, 26, 30, 32, 46]. For the Blockchain use-case, the granularity is a block of transactions. Thus, unlike the general setting, Block-STM do not need to handle a long-lived stream of transactions that arrive at arbitrary times and commit them one by one. Moreover, thanks to the VM, the Blockchain use-case does not require opacity [24].

Preset and deterministic order. There is prior work on designing STM libraries constrained to the predefined serialization order [34, 42, 50]. In [34, 50] each transaction is committed by a designated thread and thus the predefined order reduces resource utilization. This is because threads have to stall until all previous transactions in the order are committed before they can commit their own. Transactions in [42] are also committed by designated threads, but they limit the stalling periods to only the latency of the commit via a complex forwarding locking mechanism and flat combining [27] based validation.

Deterministic STM libraries [36, 38, 49, 52] consider a less restricted case in which every execution of the same set of transaction produces the same final state. The idea in the state-of-the-art [52] is simple. All transactions are executed from the initial state and the maximum independent set of transaction (i.e., with no conflicts among them) is committed, arriving to a new state. The remaining transaction are executed from the new state, the maximum independent set is committed, and so on. This approach thrives for low conflict workloads, but otherwise suffers from high overhead.

To summarize, in the context of STM literature, the (deterministic or preset) ordering constraints have been viewed as a “curse”, i.e. an extra requirement that the system needs to satisfy at the cost of added overhead. For the Block-STM approach, on the other hand, the preset order is the “blessing” that the whole algorithm is centered around. In fact, the closest works to Block-STM in terms of how the preset serialization order is used to deal with conflicts are from the databases literature. Calvin [48] and BOHM [21] use batches (akin to blocks) of transactions and their preset order to execute transactions when their read dependencies are resolved. This is possible because, in the databases context, the write-sets of transactions are assumed to be known in advance. This assumption is not suitable for Blockchains as smart contracts might encode an arbitrary logic. Therefore, Block-STM does not require the write-set to be known and learns dependencies on the fly.

Multi-version data-structures. Multi-version data structures are designed to avoid write conflicts [10]. They have a history of applications in the STM context [13, 37], some of which utilize optimistic concurrency control [12]. The multi-version data-structure maps between memory locations and

values that are indexed based on versions that are assigned to transactions via global version clock [12, 17, 39].

Blockchain execution. The connection between STM techniques and parallel smart contract execution was explored in the past [5, 7, 8, 18]. A *miner-replay* paradigm was explored in [18], where miners parallelize block execution using a white-box STM library application that extracts the resulting serialization order as a “fork-join” schedule. This schedule is sent alongside the new block proposal (via the consensus component) from miners to validators. After the block is proposed, validators utilize the fork-join schedule to deterministically replay the block. ParBlockchain [5] introduced an *order-execute* paradigm (OXII) for deterministic parallelism. The ordering stage is similar to the schedule preparation in [18], but the transaction dependency graph is computed without executing the block. OXII relies on read-write set being known in advance via static-analysis or on speculative pre-execution to generate the dependency graph among transactions. OptSmart [7, 8] makes two improvements. First, the dependency graph is compressed to contain only transactions with dependencies; those that are not included may execute in parallel. Second, execution uses multi-versioned memory to mitigate write-write conflicts.

Hyperledger Fabric [6] and several related works [41, 43] follow the execute-order-validate paradigm. As a result, the execution phase can abort unserializable transactions before ordering. Transactions in [15] are pre-executed off the critical path to produce hints for final execution.

6 Summary

This paper presents Block-STM, a parallel execution engine for the Blockchain use-case that achieves up to 170k tps with 32 threads in our benchmarks. For a fully sequential workload, it has a smaller than 30% overhead, mitigating any potential performance attacks. Block-STM relies on the write-sets of transactions’ last incarnations to estimate dependencies and reduce wasted work. If write-set pre-estimation was available, e.g., with a best effort static analysis, it could be similarly used by the first incarnation of a transaction. Moreover, using static analysis to find the best preset order is an interesting future direction.

Block-STM uses locking for synchronization in the [Scheduler](#) module. It is possible to use standard multicore techniques to avoid using locks, however, we did not see significant performance difference in our experiments. Thus, we chose the design with locks for the ease of presentation.

In Blockchain systems, there is usually an associated “gas” cost to executing transactions. A single location for gas updates, could make any block inherently sequential. However, this issue is typically avoided by tracking gas natively, burning it or having specialized types or sharded implementation.

As discussed in the Section 4, Diem VM currently does not support suspending and resuming transaction execution.

Once this feature is available, Block-STM can restart execution from the read that caused suspension upon encountering a dependency. A potential optimization to go along with this feature is to validate the reads that happened during the execution prefix (before transaction was suspended) upon resumption. This could allow earlier detection of impending aborts.

The current Block-STM implementation is not optimized for NUMA architectures or hyperthreading. Exploring these optimizations is another direction for future research. Another interesting direction is to explore nesting techniques [35] for transactional smart contract design.

Acknowledgment

The authors would like to thank Sam Blackshear and Avery Ching for fruitful discussions.

References

- [1] [n.d.]. Aptos codebase. <https://github.com/aptos-labs/aptos-core>.
- [2] [n.d.]. Aptos whitepaper. <https://github.com/aptos-labs/aptos-core/blob/main/developer-docs-site/static/papers/whitepaper.pdf>.
- [3] [n.d.]. Diem codebase. <https://github.com/diem/diem/tree/main>.
- [4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 11–20.
- [5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In *proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1337–1347. <https://doi.org/doi:10.1109/ICDCS.2019.00134>
- [6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
- [7] Parwat Singh Anjana, Hagit Attiya, Sweta Kumari, Sathya Peri, and Archit Somani. 2020. Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory. In *International Conference on Networked Systems*. Springer, 77–93.
- [8] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2021. OptSmart: A Space Efficient Optimistic Concurrent Execution of Smart Contracts. [arXiv:2102.04875 \[cs.DC\]](https://arxiv.org/abs/2102.04875)
- [9] Hagit Attiya and Jennifer Welch. 2004. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons.
- [10] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.
- [11] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. *Libra Assoc.* (2019).
- [12] Edward Bortnikov, Eshkar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. 2017. Omid, Reloaded: Scalable and {Highly-Available} Transaction Processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 167–180.
- [13] Joao Cachopo and António Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. *Science of Computer Programming* 63, 2 (2006), 172–185.
- [14] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Commun. ACM* 51, 11 (2008), 40–46.
- [15] Yang Chen, Zhongxin Guo, Runhui Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 570–587.
- [16] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2019. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. [arXiv preprint arXiv:1904.05234](https://arxiv.org/abs/1904.05234) (2019).
- [17] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.
- [18] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2020 (ArXiv version 2017). Adding concurrency to smart contracts. *Distributed Computing* 33, 3 (2020 (ArXiv version 2017)), 209–225.
- [19] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2011. Why STM can be more than a research toy. *Commun. ACM* 54, 4 (2011), 70–77.
- [20] Avner Elizarov, Guy Golan-Gueta, and Erez Petrank. 2019. LOFT: lock-free transactional data structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 425–426.
- [21] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multi-version concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.
- [22] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 237–246.
- [23] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. 2022. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. [arXiv preprint arXiv:2203.06871](https://arxiv.org/abs/2203.06871) (2022).
- [24] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 175–184.
- [25] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2006. *Stmbench7: a benchmark for software transactional memory*. Technical Report.
- [26] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. Optimistic transactional boosting. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 387–388.
- [27] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 355–364.
- [28] Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 207–216.
- [29] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*. 289–300.
- [30] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.
- [31] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.
- [32] Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. 2019. Wait-free dynamic transactions for linked data structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. 41–50.
- [33] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, Vol. 509518.
- [34] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM Sigplan Notices* 44, 6 (2009), 166–176.
- [35] John Eliot Blakeslee Moss. 1981. *Nested Transactions: An Approach to Reliable Distributed Computing*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.
- [36] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, portable and parameterless. *ACM SIGPLAN Notices* 49, 4 (2014), 499–512.
- [37] Dmitri Perelman, Rui Fan, and Idit Keidar. 2010. On maintaining multiple versions in STM. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 16–25.

- [38] Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. 2014. DeSTM: harnessing determinism in STMs for application development. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 213–224.
- [39] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*. Springer, 284–298.
- [40] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Multi-queues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 80–82.
- [41] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 543–557.
- [42] Mohamed M Saad, Masoomeh Javidi Kishi, Shihao Jing, Sandeep Hans, and Roberto Palmieri. 2019. Processing transactions in a predefined order. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 120–132.
- [43] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*. 105–122.
- [44] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.
- [45] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 68–70.
- [46] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional data structure libraries. *ACM SIGPLAN Notices* 51, 6 (2016), 682–696.
- [47] The DiemBFT Team. 2021. State machine replication in the Diem Blockchain. <https://developers.diem.com/docs/technical-papers/state-machine-replication-paper>.
- [48] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*.
- [49] Tiago M Vale, João A Silva, Ricardo J Dias, and João M Lourenço. 2016. Pot: Deterministic transactional execution. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–24.
- [50] Christoph Von Praun, Luis Ceze, and Calin Caşcaval. 2007. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 79–89.
- [51] Maximilian Wohrer and Uwe Zdun. 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2–8.
- [52] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. 2019. LiTM: A Lightweight Deterministic Software Transactional Memory System. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. 1–10.

A Correctness

We consider concurrent runs⁵ by threads, where each thread performs a sequence of atomic operations, and there is a global order in which these operations appear to take place. We use the term *time* to refer to a point in this global order, i.e. a time T determines for each thread the operations that it performed before T .

A.1 Life of a Version

We say that validation of version $v = (j, i)$ starts anytime a validation task with version v is returned to some thread t , either in Line 5 or in Line 9. We say execution of version v starts immediately after Line 114 is performed that sets the status of transaction tx_j to EXECUTING(i). We say that the execution of version v *aborts* immediately after Line 151 is performed, and that the validation of version v *aborts* immediately after Line 179 is performed. In both cases, the transaction status is set to ABORTING(i).

After thread t starts the execution of version v , an execution task with v is returned either in Line 7 or in Line 9. Thread t then invokes the `try_execute` function for the execution task, which may invoke `finish_execution` procedure in Line 19. The `finish_execution` function is not called only when the execution aborts, in which case we say the execution *finishes* at the same time when it aborts. Similarly, after a validation starts, t invokes `needs_reexecution` function for the validation task, which always invokes `finish_validation` procedure in Line 26.

If Line 174 (for execution) or Line 190 (for validation) is performed, then the corresponding validation or execution *finishes* immediately before. If these lines are not performed in `finish_execution` and in `finish_validation`, respectively, then the `finish_execution` invocation returns a validation task and the `finish_validation` invocation returns an execution task. We say that such an execution *finishes* immediately before the `try_execute` invocation returns in Line 5 (i.e. before validation starts for the version in the returned task). Analogously, such a validation *finishes* immediately before a `needs_reexecution` invocation returns in Line 7 (i.e. before execution starts for the version in the returned task).

An update to a transaction status is always performed by a thread while holding the corresponding lock. Figure 2 describes all possible status transitions. For example, once `txn_status[j]` becomes EXECUTING(i), it can never be READY_TO_EXECUTE(i) at a later time. By the code, illustrated in the allowable transitions in Figure 2, we have

Corollary 1. *The following observations are true:*

- *The status of transaction tx_j must be set to READY_TO_EXECUTE(i) in Line 158 before the execution of the version $v = (j, i)$ can start.*

⁵Typically called executions in the literature, but we use the term *run* to avoid a naming clash with transaction execution.

- *Any version $v = (j, i)$ can be executed at most once (by a thread that updates the status of transaction tx_j to EXECUTING(i) from READY_TO_EXECUTE(i) to start the execution of v). Only the executing thread may update the status next, either to ABORTING(i) in Line 151 or to EXECUTED(i) in Line 166.*
- *The status of transaction tx_j must be set to EXECUTED(i) in Line 166 during the execution of version $v = (j, i)$ before any validation of v can start. Once the status is set to EXECUTED(i), it can only be updated to ABORTING(i) in Line 179 during a validation of v .*
- *At most one execution or validation of version $v = (j, i)$ can abort, updating the status to ABORTING(i) either in Line 151 from EXECUTING(i) or in Line 179 from EXECUTED(i). The next update to the status of transaction tx_j must be to READY_TO_EXECUTE($i + 1$) in Line 158.*

A.2 Safety

We say that a *pre-validation* of transaction tx_j starts any time some thread t performs a *fetch_and_increment* operation, returning j , in Line 130. The pre-validation finishes immediately before t performs Line 135, if this line is performed. Otherwise, by code, a validation task for transaction tx_j is returned from the `next_version_to_validate` function invocation. In this case, pre-validation finishes immediately before the validation task is returned in Line 9, i.e. before the corresponding validation starts.

Definition 1 (Global Commit Index). *The global commit index at time T is defined as the minimum among all the following quantities at time T :*

- `Scheduler.validation_idx`
- *all indices j , such that `Scheduler.txn_status[j].status` ≠ EXECUTED*
- *transaction indices with ongoing pre-validation*
- *transaction indices of versions with ongoing execution or validation*

We say that transactions tx_0, \dots, tx_k of the block are *globally committed* at time T if the global commit index at time T is strictly greater than k . Next, we prove the essential properties of the commit definition.

Claim 1. *If transaction tx_k is committed at time T , then it is also committed at all times $T' \geq T$.*

Proof. We prove this claim by a simple inductive reasoning on time. Specifically, for every time $T' \geq T$ we prove that k is strictly less than the global commit index at time T' . The base case for time T follows from the Claim assumption. For the inductive step, we suppose the assumption holds at time T' and show that the Definition 1 still leads to a global commit index $> k$ when the next event after T' takes effect.

- The operation may change validation index from time T' only in Line 104, which can be due to a call in Line 171

(during `finish_execution`) or in Line 185 (during `finish_validation`). In the first case, if `validation_idx` is reduced to value j , there must be an ongoing execution with transaction index j at time T' . In the second case, there must be an ongoing validation with transaction index j at time T' . Thus, in both cases, by inductive hypothesis, $j > k$.

- The operation may change a status of transaction tx_j from EXECUTED only in Line 179, in which case there is an ongoing validation with transaction index j at time T' . Thus, by inductive hypothesis, $j > k$.
- A *fetch-and-increment* operation in Line 130 may start a pre-validation of transaction tx_j . The `validation_idx` must have been j at time T' and by inductive hypothesis, $j > k$.
- If validation of a version v with transaction index j starts immediately after T' , then there must have been a pre-validation or an execution of version v that ended immediately before, hence, that was ongoing at time T' . Thus, by inductive hypothesis, $j > k$.
- If an execution of a version v with transaction index j starts immediately after T' , then let us consider two cases:
 - if an execution task was returned in Line 7, then there was a validation of a version with index j (previous incarnation) that ended immediately before, and hence, was ongoing at time T' . Thus, by inductive hypothesis, $j > k$.
 - if an execution task was returned to some thread t in Line 9, then, by the code, the status of transaction tx_j must have been previously set to EXECUTING by t . By Corollary 1, the status of transaction tx_j may not change to EXECUTED until t starts the execution. Thus, since the status of transaction tx_j is not EXECUTED at time T' , by inductive hypothesis, $j > k$.

Hence, the global commit index is monotonically non-decreasing with time. \square

Next, we prove some auxiliary claims regarding the interplay between transaction status and shared (execution and validation) indices.

Claim 2. *Suppose all transactions are eventually committed, and that at all times after T the status of transaction tx_j is EXECUTED. If no validation of a version of tx_j starts after T , then the validation index must be $> j$ at all times after T .*

Proof. Let us assume for contradiction that `validation_idx` is at most j at some time $T' \geq T$. Since all transactions are eventually committed and due to Claim 1, `validation_idx` must have value `BLOCK.size() > j` at some time after T' . The validation index is only incremented in Line 130, which is by definition a start of pre-validation. Therefore, transaction tx_j must start pre-validation after T' , and pre-validation must

finish due to Definition 1 since all transactions are eventually committed. By the claim assumption, transaction tx_j 's status is EXECUTED, so by code (due to Line 133), pre-validation finish must lead to a start of a validation of a version of tx_j , giving the desired contradiction. \square

Claim 3. *Suppose all transactions are eventually committed, and i is the highest incarnation of transaction tx_j such that version $v = (j, i)$ is executed. Then, v must start validation after Line 166 is performed in the execution of v .*

Proof. The execution of version v sets the status of transaction tx_j to EXECUTED(i) in Line 166. The execution of v eventually finishes due to Definition 1 and Claim 1, as transaction tx_j eventually commits. If a validation task is returned in Line 173, then a validation of version v starts immediately after execution finishes. Otherwise, by Corollary 1, the status of transaction tx_j will remain EXECUTED(i) unless it is updated to ABORTING(i) by some validation of v , which also concludes the proof of the claim. If the status remains EXECUTED(i) and a validation task is not returned, then validation index has a value at most j after the status update in Line 166 due to Line 169 and Line 171. Then, Claim 2 implies that a validation must start after Line 166 is performed. \square

Next, we establish the correctness invariant of the committed transactions. When we refer to a *sequential run* of all transactions, we mean the execution of transaction tx_0 , followed by the execution of transaction tx_1 , etc, for all transactions in the block.

Lemma 1. *After all transactions are committed, `MVMemory` contains exactly the paths written in the sequential run of all transactions. Moreover, a read of a path from `MVMemory` with `txn_idx = BLOCK.size()` returns the same value as the contents of the path after the sequential run.*

Proof. Suppose all transactions are eventually committed. Since initial status for each transaction is READY_TO_EXECUTE, while Definition 1 requires status EXECUTED, by the code, for each transaction tx_j the version $(j, 0)$ must start executing at some point. Also, due to the commit definition and Claim 1, all executions that start must finish (in order for the transactions to eventually be committed). In fact, by Claim 1 the total number of executions, validations and pre-validations must be finite and they must all finish. For each transaction index j , let m_j be the highest incarnation for which there is an execution of version (j, m_j) . By Corollary 1, among the versions of transaction tx_j that are executed, version (j, m_j) is executed last. We show by induction on j that the execution of version (j, m_j) reads the same paths and values from `MVMemory` as the execution of transaction tx_j would during the sequential run. Thus, at the end of version (j, m_j) execution, all entries with transaction index j in `MVMemory` also correspond to the same paths and contain the same values as the write-set in the sequential run.

The base case holds because every read with $txn_idx = 0$ reads from storage. Next, suppose the inductive claim holds for transactions tx_0, \dots, tx_k . By Claim 3, version $v_{k+1} = (k+1, m_{k+1})$ is validated at least once after Line 166 is performed during v_{k+1} 's (unique, by Corollary 1) execution. Any validation of v_{k+1} that starts also finishes in order for the global commit index to reach values above $k+1$. Finally, no validation of version v_{k+1} may abort, as this would set $txn_status[k+1]$ to an ABORTING status and prevent global commit index from reaching `BLOCK.size()` without another incarnation of transaction tx_{k+1} , contradicting the maximality of m_{k+1} . Therefore, we only need to show that a value read at any access path during the validation of v_{k+1} is the same as in the sequential run of transaction tx_{k+1} . Then, since the validation must succeed, the execution of v_{k+1} must have read the same values, and produced a compatible output to the sequential run, proving the inductive step.

Let α be the validation of v_{k+1} that starts last. Let p be any path read during α , and let v_p be the corresponding version observed during the `validate_read_set` invocation that returned `true` (if the read returned a `READ_ERROR` in Line 67 then α would fail). If $v_p = \perp$, then validation α , and the corresponding execution of version v_{k+1} both read from storage. If v_p is a version of some transaction tx_j , since `MVMemory` only reads values from lower transactions, we have $j < k+1$. Version v_p is written during a record call invoked in Line 18 during an execution that sets the status of transaction tx_j to an EXECUTED status before finishing. We show this must have been the last execution of tx_j using a proof by contradiction. Otherwise, by Corollary 1, a validation β of the same version of tx_j must follow and abort. Thus, by code, before finishing, β marks path p as an ESTIMATE, after it is read by α . The `validation_idx` is then ensured to be at most j in Line 185 in β , contradicting Claim 2 (Due to Claim 3 the status of transaction tx_{k+1} is set to EXECUTED(m_{k+1}) in Line 166 during the execution of v_{k+1} , before α starts. Since no validation of v_{k+1} aborts, by Corollary 1, $txn_status[k+1]$ never changes from EXECUTED).

Hence, if v_p is a version of tx_j , then the value read from p is in fact the value written at path p during the execution of the last tx_j 's version (j, m_j) . By the induction hypothesis, this is the same value that transaction tx_j writes at p in the fully sequential run. To finish the proof, suppose for contradiction that in the sequential run transaction tx_{k+1} reads a value written by transaction $tx_{j'}$ with $j' > j$. The validation α did not observe any entry from j' at path p , not even an ESTIMATE. However, by induction hypothesis, during the execution of version $(j', m_{j'})$ the same value as in the sequential run must be written to path p . Therefore, after a read by α , there is an execution of a version of transaction $tx_{j'}$ that sets `wrote_new_path` to `true` due to p and decreases validation index by calling Line 171. This again contradicts our assumption about α and completes the proof, as the

argument when $v_p = \perp$ instead of $v_p = (j, m_j)$ is analogous. \square

A.2.1 Number of Active Tasks. What is left is to show is the safety of the `check_done` mechanism for determining when the transactions are committed. The key is to understand the role of the `num_active_tasks` variable in the `Scheduler` module. The `num_active_tasks` is initialized to 0 and incremented in Line 122 and Line 129. The increment in Line 129 is accounting for the pre-validation that starts with a `fetch-and-increment` in the following line (Line 130). The `num_active_tasks` is decremented in Line 135 if no validation task corresponding to the fetched index is created. Otherwise, pre-validation leads to the start of a validation, and `num_active_tasks` is decremented immediately after the validation finishes, in Line 190 (unless an execution task is created for the caller). The logic for execution tasks is analogous, with one difference that an execution can also finish in Line 151, in which case `num_active_tasks` is decremented shortly after, in Line 153. When `finish_execution` or `finish_validation` functions return a new task to the caller, `num_active_tasks` is left unchanged (instead of incrementing and decrementing that cancel out). It follows that `num_active_tasks` is always non-negative. The following auxiliary claims establish useful properties of when the value becomes 0.

Claim 4. *Suppose the status of transaction tx_j was set to `READY_TO_EXECUTE` at time T , and did not change until a later time T' . If execution index was at most j at some time between T and T' , then either `num_active_tasks` > 0 or `execution_idx` $\leq j$ at time T' .*

Proof. Let us assume for contradiction that at time T' `num_active_tasks` is 0 and `execution_idx` is strictly larger than j , but that at some time between T and T' , the execution index was at most j . Since execution index reaches a value larger than j by time T' , a `fetch-and-increment` operation must have been performed in Line 123 between T and T' , returning j . The `num_active_tasks` counter is incremented in the previous line, in Line 122 (this is very similar to the increment to account for pre-validation, while here it is an analogous pre-execution stage). Since the status is of transaction tx_j remains `READY_TO_EXECUTE` until T' , the only way to reduce `num_active_tasks` to 0 at time T' is to perform the corresponding decrement, which by code, would occur only after an execution of a version of transaction tx_j (due to Line 113). However, before an execution finishes (and then `num_active_tasks` is decremented), it must perform Line 113 and since the status of transaction tx_j is `READY_TO_EXECUTE`, it must update the status to `EXECUTING` in Line 114, giving the desired contradiction with assumption in the claim. \square

Lemma 2. *Suppose $execution_idx \geq BLOCK.size()$, $validation_idx \geq BLOCK.size()$ and num_active_tasks is 0 simultaneously at time T . Then, all transactions are committed at time T .*

Proof. As num_active_tasks is 0 at time T , there may not be an ongoing pre-validation, validation or execution at time T . This is because an increment corresponding of num_active_tasks always occurs before the start, while the decrement always occurs after the finish of the corresponding pre-validation, validation or execution. Next, we will prove that for any transaction index j , `Scheduler.txn_status[j].status = EXECUTED` at time T . Then, by Definition 1, the global commit index is equal to the $validation_index$, which is at least $BLOCK.size()$, meaning that all transactions are committed at time T .

In the following, we prove by contradiction that all transactions must have an EXECUTED status at time T . Suppose j is the smallest index of a transaction with a non-EXECUTED status. Consider three cases:

- `Scheduler.txn_status[j].status = READY_TO_EXECUTE`. We consider the time when the READY_TO_EXECUTE status was last set for transaction tx_j in Line 113. This is due to a call either in Line 161 or in Line 184.
 - Call in Line 161: there is an ongoing execution, which must finish in order for num_active_tasks to be 0 at time T . Before finishing, the `decrease_execution_idx` invocation in Line 164 ensures that the execution index has a value at most j . Thus, by Claim 4, the execution index is at most j at time T . A contradiction.
 - Call in Line 184: there is an ongoing validation which must finish in order for num_active_tasks to be 0 at time T . Before finishing, $execution_idx$ must be observed in Line 186 to be strictly higher than j , or we would get a contradiction with Claim 4. But then, `try_incarnate` must be called in Line 187, which by code, would observe READY_TO_EXECUTE status and update it to EXECUTING, contradicting the status at time T .
- `Scheduler.txn_status[j].status = EXECUTING`. By Corollary 1 and the definition of execution, there must be an ongoing execution at time T (of a version of tx_j by the thread that set the status), which we already showed is impossible.
- `Scheduler.txn_status[j].status = ABORTING`. Let T' be the time when the ABORTING status was last set for transaction tx_j , which can be in Line 151 or in Line 179.
 - call in Line 151 in an `add_dependency` invocation: in this case, txn_idx must be j and the thread must be holding a lock on the status of a `blocking_txn_idx`, which we will call j' . Because `MVMemory` only reads entries, including an ESTIMATE, from lower transactions, and reading an ESTIMATE is required for calling the `add_dependency` function, we have

$j' < j$. Since Line 151 was performed, due to the check in Line 149, the status of transaction $tx_{j'}$ cannot be EXECUTED, but by the minimality of tx_j it must be EXECUTED at time T . Therefore, an execution of a version of $tx_{j'}$ must invoke Line 166 between times T' and T . This execution must finish in order for num_active_tasks to be 0 at time T , meaning that `resume_dependencies` invocation in Line 168 must be completed before T . However, due to locks, tx_j is now a dependency of $tx_{j'}$, and this `resume_dependencies` invocation must update the status of transaction tx_j to READY_TO_EXECUTE due to the call in Line 161, contradicting the status at time T .

- call in Line 179: there is an ongoing validation which must finish in order for num_active_tasks to be 0 at time T . Before finishing, the status must be updated to READY_TO_EXECUTE due to the call in Line 184, contradicting the status at time T . \square

A.2.2 Safety Guarantees.

Lemma 3. *Let time T be right before the operation in Line 99 or operation in Line 104 by thread t takes effect. Suppose num_active_tasks is 0 at some time $T' \geq T$. Then, thread t must have incremented `decrease_cnt` (in Line 100 or in Line 105) between times T and T' .*

Proof. Performing Line 99 as a part of `decrease_execution_idx` reduces $execution_idx$ to the minimum of $execution_idx$ and $target_idx$, while performing Line 104 as a part of `decrease_validation_idx` is similar for the $validation_idx$. The `decrease_execution_idx` procedure is invoked only in Line 164 as a part of an ongoing execution, and accounting for this execution, num_active_task must be at least 1 during the whole invocation. Hence, in order for num_active_tasks to become 0, it must be decremented after the execution completes. Thus, t must first complete `decrease_execution_idx`, which includes performing Line 100.

The `decrease_validation_idx` procedure is invoked either as a part of validation that aborts, or as a part of execution when `wrote_new_path` is `true` in `finish_execution`. In both cases, num_active_tasks is at least 1 accounting for the ongoing validation or execution, since both finish after `decrease_validation_idx` invocation completes. Hence, in order for num_active_tasks to become 0, by code, t must decrement it after it finishes execution of validation. However, before doing so, it must perform Line 105 and return from the `decrease_validation_idx` invocation. \square

Theorem 1. *If a thread joins after invoking the `run` procedure, then all transactions are necessarily committed at that time.*

Proof. The threads return from the `run` invocation when they observe a `done_marker = true` in Line 102. The `done_marker`

is set to *true* in Line 109 after observing that *validation_idx* \geq BLOCK_SIZE, *execution_idx* \geq BLOCK_SIZE and *num_active_tasks* is 0. These checks are not performed atomically, but instead a double-collect mechanism is used on the *decrease_count* variable, which is a monotonically non-decreasing counter. In particular, *check_done* confirms that *decrease_count* did not change (increase) while *execution_idx*, *validation_index* and *num_active_tasks* were read.

Since a thread joined, *decrease_count* did not increase while it first observed *execution_idx* to be at least BLOCK_SIZE at time T_1 , then observed *validation_idx* to be at least BLOCK_SIZE at time $T_2 > T_1$, and finally observed *num_active_tasks* to be 0 at time $T_3 > T_2$. We show by contradiction that *num_active_tasks* was 0 and *execution_idx* and *validation_idx* were still at least BLOCK.size() simultaneously at T_3 . Assume by contradiction that T_3 does not have this property. Thus, *execution_idx* must be decreased between T_1 and T_3 or *validation_idx* must be decreased between T_2 and T_3 . In both cases, we can apply Lemma 3, implying that *decrease_count* must have been incremented between T_1 and T_3 , giving the desired contradiction.

Therefore, *check_done* only succeeds if the number of active tasks is 0 while the execution index and the validation index are both at least BLOCK.size() at the same time. By Lemma 2 and, all transactions must be committed at this time. \square

The *MVMemory.snapshot* function internally calls *read* with *txn_id* = BLOCK.size() for all affected paths. By Theorem 1 all transactions are committed after a thread joins, so Lemma 1 implies the following

Corollary 2. *A call to *MVMemory.snapshot()* after a thread joins returns the exact same values at exact same paths as would be persisted at the end of a sequential run of all transactions.*

A.3 Liveness

We prove liveness under the assumption that every thread keeps taking steps until it joins⁶ and that the *VM.execute* is wait-free. We start by formally defining *pre-execution* in an analogous fashion to pre-validation. A pre-execution of a transaction tx_j starts any time some thread t performs a *fetch_and_increment* operation, returning j , in Line 123. The pre-execution finishes immediately before t performs Line 116, if this line is performed. Otherwise, by code, an execution task for transaction tx_j is returned from the *next_version_to_execute* function invocation. In this case, pre-execution finishes immediately before the execution task is returned in Line 9, i.e. before the corresponding execution starts.

⁶A standard assumption used to prove deadlock-freedom and starvation-freedom of algorithms, which are equivalent in our, single-shot, setting.

Lemma 4. *There are finitely many pre-executions, executions, pre-validations and validations.*

Proof. We prove the lemma by induction on transaction index, with a trivial base case (no pre-execution, execution, pre-validation or validation occurs for transactions with indices < 0). For the inductive step, show that for any transaction index k there are finitely many associated pre-executions, pre-validations, executions or validations. For the inductive hypothesis, we only assume that there are finitely many executions and validations for versions of transactions indexed $< k$. It implies that after some finite time T :

- (a) the execution index is never updated to a value $\leq k$ in Line 99. The *decrease_execution_idx* procedure is only called in Line 164 as a part of an ongoing execution of some transaction tx_j when execution index is reduced to the minimum index of other transactions that depend on tx_j , all of which must have index $> j$ (as only higher-indexed transactions could have read from *MVMemory* an ESTIMATE written during tx_j 's execution and become a dependency).
- (b) the entries in *MVMemory* for transactions indexed lower than k never change. This holds because *MVMemory.record* invocation in Line 18 that affects entries with transaction index j , is, as defined, a part of transaction tx_j 's execution.

Due to (a), only one pre-execution of transaction tx_k may start after time T , so there are finitely many pre-executions for tx_k in total. Next, we show that there is at most one validation of a version of tx_k that aborts after time T . If such a version exists, let (k, i) be the first version that aborts after T . Due to Corollary 1, version (k, i) may not abort more than once, and after it aborts, an execution of version $(k, i+1)$ must complete before any validation of version $(k, i+1)$ (or higher) starts. However, no validation of version $(k, i+1)$ may abort, since by (b), the entries associated with transaction indices strictly smaller than k no longer change in the multi-version data-structure, i.e. *MVMemory.validate_read_set* for a version whose execution started after T necessarily returns *true* in Line 22. Thus, after some finite time no execution of a version of transaction tx_k may start, as this only happens either following a pre-execution or a validation that aborts. Moreover, we can now show that similar to (a) for the execution index, after some finite time, the validation index can never be reduced to a value $\leq k$ in Line 104. This is because the *decrease_validation_idx* procedure is either called in Line 171, when the *validation_idx* may be reduced to j as a part of a transaction tx_j 's ongoing execution, or it is called in Line 185, when the validation index may be reduced to $j+1$ as a part of a transaction tx_j 's ongoing validation.

Therefore, there are finitely many pre-validations of transaction tx_k and as a result, no validation of a version of tx_k may start after some finite time. This is because a validation starts either following a pre-validation, or an execution of

a version of tx_k . As there are finitely many threads, we obtain that there are finitely many total pre-validations and pre-executions of transaction tx_k , as well as executions and validations versions of tx_k . \square

In Block-STM, locks are used to protect statuses and dependencies for transactions. We now prove starvation-freedom for these locks.

Claim 5. *If threads keep taking steps before they join, then any thread that keeps trying to acquire a lock eventually succeeds.*

Proof. A lock on transaction dependencies is acquired in Line 148 or in Line 167, both of which, by definition, occur as a part of some version's execution. There are more cases of when a lock on a transaction status may be acquired. The operations in Line 112 and in Line 132 are a part of a pre-execution of pre-validation of some transaction, respectively. The lock may be acquired in Line 156 in order to set the `READY_TO_EXECUTE` status as a part of an ongoing execution (call to `set_ready_status` in Line 161) or validation (call in Line 184). The operation in Line 166 sets the status to `EXECUTED` as a part of an ongoing execution, and the operation in Line 179 sets the status to `ABORTING` as a part of an ongoing validation (that aborts). The remaining two instances in Line 149 and in Line 151 occur as a part of a version's execution when a dependency is encountered, while the thread is also holding a lock on dependencies. These are the only instances when a thread may simultaneously hold more than one lock, and also only the two operations within any critical section that may involve waiting. Because the acquisition order in these cases is unique (first the lock for dependencies, then for status) and all threads keep taking steps, a deadlock is therefore impossible.

Moreover, as described above, all acquisitions happen as a part of an ongoing pre-execution, pre-validation, execution or validation. By Lemma 4, there are finite number of these, implying that in our setting, deadlock-freedom is equivalent to starvation-freedom, i.e. as long as threads keep taking steps, any thread that tries to acquire a lock in Block-STM must eventually succeed. \square

Combining the above claims, we show

Corollary 3. *Suppose all threads keep taking steps before they join and `VM.execute` is wait-free. Then, after some finite time, there may not be any ongoing pre-execution, pre-validation, execution or validation.*

Proof. By Lemma 4, there are finitely many pre-executions, pre-validations, executions and validations. Since all threads keep taking steps, to complete the proof we need to show that they all finish within finitely many steps of the invoking thread. This is true because `VM.execute` is assumed to be wait-free, lock are acquired within finitely many steps

by Claim 5, and by code there is no other potential waiting involved in pre-execution, pre-validation, execution or validation. \square

Theorem 2. *If threads keep taking steps before they join and `VM.execute` is wait-free, then all threads eventually join.*

Proof. For contradiction, suppose some thread never joins. By the theorem assumption, the thread keeps taking steps and by Claim 5, it acquires all required locks within finitely many steps. Moreover, since the `VM.execute` function is wait-free, by Corollary 3, after some finite time there can be no ongoing pre-execution, pre-validation, execution or validation. By code, the thread in this case must keep repeatedly entering the loop in Line 3 and invoking `next_task` in Line 9, while both the execution index and the validation index are always \geq `BLOCK.size` - otherwise, a pre-execution or pre-validation would commence.

Since `decrease_execution_idx` and `decrease_validation_idx` procedures are only invoked as a part of an ongoing execution or validation, respectively, after some finite time, this counter remains unchanged. Finally, by the mechanism that counts the active tasks, described in Section A.2.1, `num_active_tasks` counts ongoing pre-executions, pre-validations, executions and validations. By code and since all threads keep taking steps before they join, the counter is always decremented after these finish. Since by Lemma 4, all pre-executions, pre-validations, executions and validations eventually finish, after some finite time the `num_active_tasks` counter must always be 0.

The thread that repeatedly invokes `next_task` must also repeatedly call `check_done` procedure. However, by the above, after some finite time it must set the `done_marker` to `true` in Line 109. However, the next time the thread reaches Line 3, it will not enter the loop and join, proving the theorem by contradiction. \square