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

CCS Concepts: • **Theory of computation → Concurrent algorithms**; • **Computing methodologies → Shared memory algorithms**.

Keywords: parallel execution, STM, blockchain

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 [18]).

Conflicts are the main challenge for performance. An approach pioneered by Software Transactional Memory (STM) libraries [32, 46] is to instrument memory accesses to detect conflicts. STM libraries with optimistic concurrency control [19] (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 [7, 10, 20] 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 [24] 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

*Rati Gelashvili, Alexander Spiegelman, and Zhuolun Xiang share first authorship.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577524

a multi-version data-structure [12], 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.

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. Anther advantage of the of the preset order is that it allows as comprehensive correctness testing as we can compare to a sequential execution output.

Three 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 [11]. Second, transactions are specified in smart contract languages, such as Move [13] and Solidity [54], and run in a virtual machine (VM) that encapsulates their execution and ensures safe behavior. Therefore, opacity [27] is not required to protect against fatal errors, allowing Block-STM to efficiently combine an optimistic concurrent control with multi-version data structure, without additional mechanisms to avoid reaching inconsistent states. Third, infinite loops, even speculatively, are not possible in the Blockchain use-case since each transaction specifies a maximum amount of 'gas' (cost) the user is willing to pay for the execution, and each operation of the transaction in the VM consumes some positive amount of gas. If all gas is consumed before execution ends, the transaction is aborted.

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 [6, 42], 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 [50] and its successor Aptos [2] open source blockchain code-bases [1, 4]. 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 [55] implementation, and performances 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 with a detailed pseudo-code deferred to the full version [26], while Section 4 describes implementation and evaluation. Section 5 discusses related work and Section 6 concludes the paper. The full version [26] also 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 $tx_1 < tx_2 < ... < tx_n$. The problem definition is to execute the block and produce the final state equivalent to the state produced by executing the transactions in sequence $tx_1, tx_2, \ldots tx_n$, 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

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

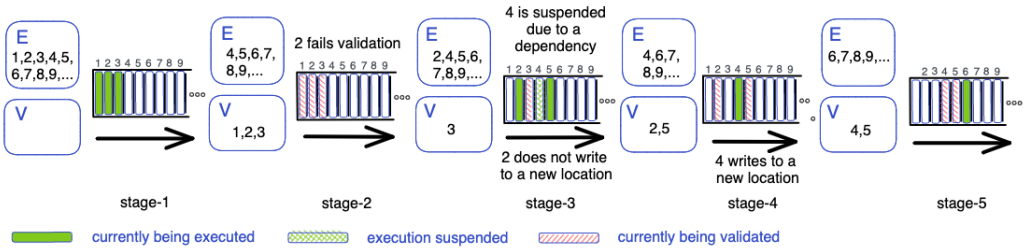

Illustration of an example execution of the abstract Block-STM collaborative scheduler.

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.

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 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 scheduler prioritizes tasks (validation and execution) associated with lower-indexed transactions. Next, we overview the high-level ideas behind the approach.

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 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). Validation failure can thus be detected without waiting for a subsequent incarnation to finish.

Commit rule. In [26], 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 $0, \ldots, j-1$ 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.

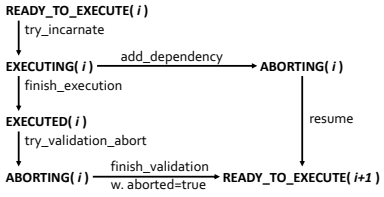

Figure 2. Illustration of status transitions

3 Block-STM Detailed Description

In this section, we describe Block-STM. Due to space constraints, we refer the full pseudo-code to the full version [26], where we also provide pseudo-code line numbers for algorithm descriptions. Upon spawning, threads perform the

run() procedure. 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. 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, and dispatches a proper handler based on its kind, i.e. function try_execute for an EXECUTION_TASK and function needs_reexecution 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.

3.1.1 Execution Tasks. An execution task is processed using the try_execute procedure. First, a VM.execute function is invoked. 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 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 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 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 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. 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.

3.1.2 Validation Tasks. A validate_read_set call 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 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, 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*) 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. Sometimes, (as an optimization), the re-execution task is returned (that proceeds to return the new version from needs_reexecution and then become the only thread to execute the next incarnation of the transaction).

3.2 Multi-Version Memory

The MVMemory module 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 $i < j$. 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 [36].

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` updates *last_written_locations* and also removes 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 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. 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, 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.

The `validate_read_set` function loads (via RCU) the most recently recorded read-set from the transaction's execution. 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, 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. We describe how reads and writes are handled in Block-STM by the VM.`execute` function (invoked while performing an execution task. 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). 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, ⊥)* 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. To elaborate on the error handling, we assume that the the virtual machine encapsulation guarantees no side effects, in particular, that no VM-internal error can automatically trigger an error outside of the VM. Instead, The VM captures errors as a part of the transaction output and returns the speculative output to BlockSTM, which validates the speculative execution as part of the algorithm. In the end, BlockSTM produces consistent outputs to the sequential execution, which also applies to error handling. In fact, any error handling based on the results of the sequential execution can be replicated by waiting for the block to be committed by Block-STM and then scanning the final outputs for errors in order.

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 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). 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). 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). 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 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]*. 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. 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, 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.

Managing 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 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_increment*-ing, 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 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. As an optimization, instead of reducing *execution_idx*, the execution task is sometimes returned to the caller.

Similarly, if a validation of transaction tx_k was successfully aborted, Block-STM ensures, in finish_validation, that *validation_idx* $\leq k$. In addition, in finish_execution of

transaction tx_k, Block-STM invokes `decrease_validation_idx` 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.

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. Threads invoke a `check_done` procedure, when observing an execution or validation index that is already $\geq$ `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 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.

The *num_active_tasks* counter is incremented, 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, or after the tasks finish. 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. 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 [3] is in Rust, and is merged on the main branch of the open source Diem and Aptos projects [2, 50]. Both Blockchains run a virtual machine for smart contracts in Move language [13]. 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.

Another related optimization implemented in Block-STM occurs when the `Scheduler.add_dependency` invocation returns *false*. This indicates that the dependency has been resolved. Instead of restarting 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.

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 transactions 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.

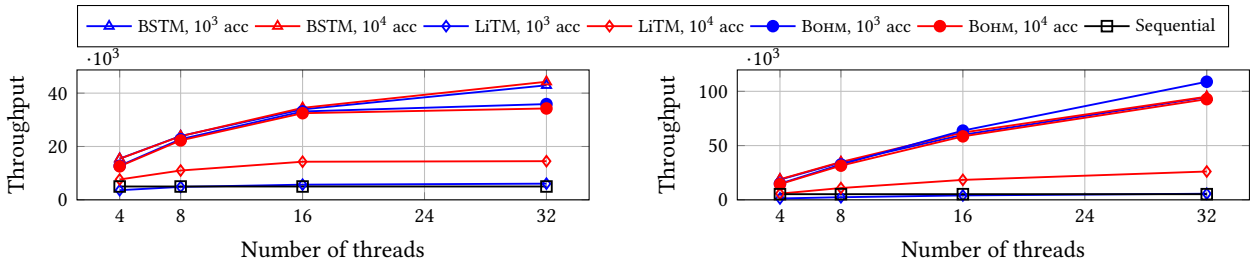

Figure 3. Comparison of BSTM, LiTM, Bohm and sequential execution for block size 10^3 (left) and 10^4 (right). Bohm is provided with perfect write estimates. Diem p2p txns.

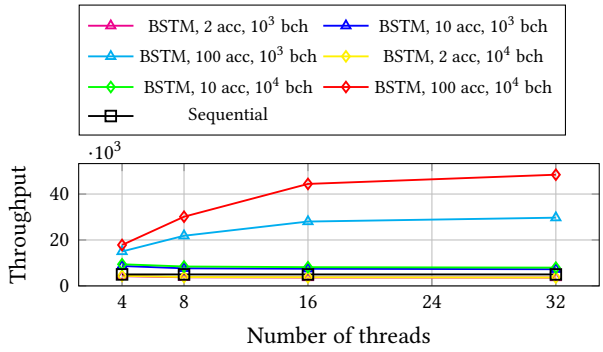

Figure 4. Comparison of BSTM and sequential execution for block size 10^3 and 10^4, account sizes 2, 10 and 100. Diem p2p transactions.

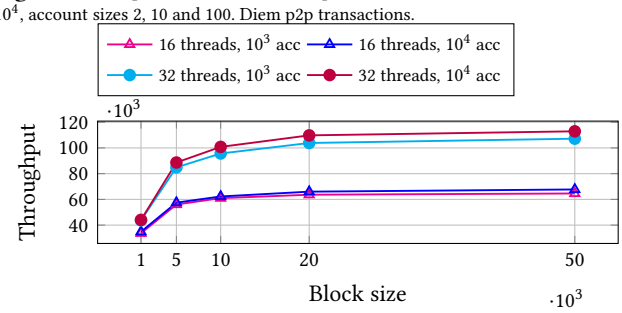

Figure 5. Throughput of BSTM for various block sizes. Diem p2p transactions.

We also perform experiments with *Aptos* p2p transactions 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 [24] and LiTM [55]. 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 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 [55], a recent deterministic STM library, claims to outperform other deterministic STM approaches on the Problem Based Benchmark Suite [47]. We describe LiTM in more detail in Section 5. In order to have a uniform setting for comparison, we implemented both a variant of Bohm and LiTM 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.

Comparison to Bohm [24]. 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.

Comparison to LiTM [55]. With 10^4 accounts, Block-STM has around 3-4x speedup over LiTM regardless of the

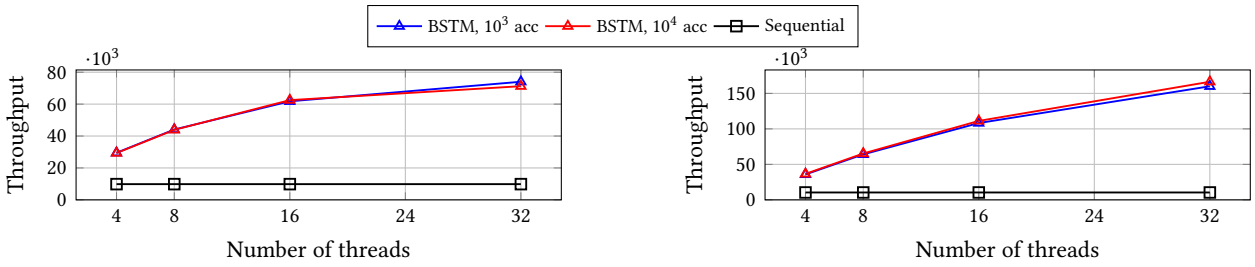

Figure 6. Comparison of BSTM and Sequential execution for block size 10^3 (left) and 10^4 (right). Aptos p2p transactions.

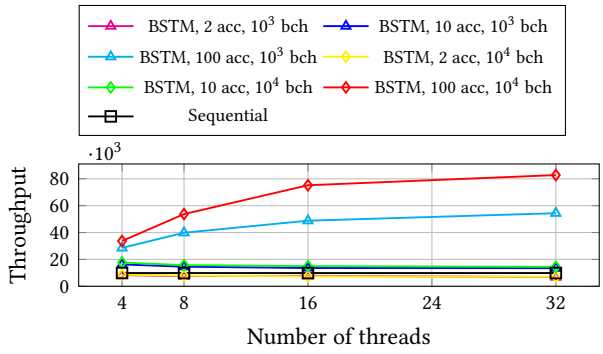

Figure 7. Comparison of BSTM and sequential execution for block size 10^3 and 10^4, account sizes 2, 10 and 100. Aptos p2p transactions.

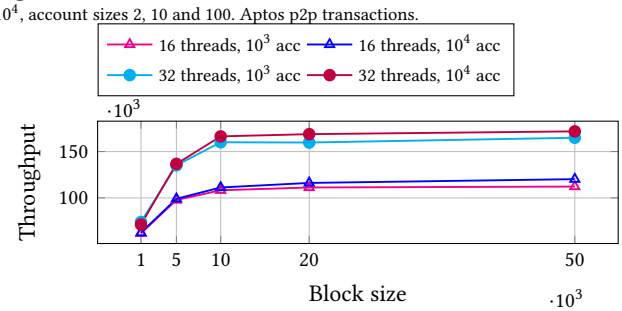

Figure 8. Throughput of BSTM for various block sizes. Aptos p2p transactions.

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., [19, 21, 25, 28, 31, 32, 46]). 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 [19, 34], 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 [38, 40, 52] 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 [16]. However, STM performance can be dramatically improved by restricting it to specific use-cases [22, 29, 33, 35, 48]. 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 [27].

Preset and deterministic order. There is prior work on designing STM libraries constrained to the predefined serialization order [37, 44, 53]. In [37, 53] 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 [44] 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 [30] based validation.

Deterministic STM libraries [38, 40, 52, 55] 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 [55] 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 [51] and Bohm [24] 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. However, as stated in Calvin, "Transactions which must perform reads in order to determine their full read/write sets (...) are not natively supported". Instead, Calvin supports a scheme called Optimistic Lock Location Prediction, modifying the transactions code to perform a pre-computation for estimating their read/write sets. These estimations might be inaccurate due to concurrency races, in which case the entire process has to be restarted. They further discuss that their approach might work for unpopular secondary indexes but struggle with popular ones. This assumption is not suitable for Blockchains as smart contracts might encode an arbitrary logic. Therefore, Block-STM does not require prior knowledge of read/write sets and learns dependencies on the fly. An unestimated write can cause transaction re-executions in Block-STM, as opposed to restarting the entire process in Calvin and Bohm.

Thread-level speculation. One setting where preset order is used with optimistic execution is thread-level speculation (TLS) [23]. This refers to a set of techniques to optimistically execute fragments of a given piece of code possibly out of order for efficiency, but guaranteeing the same output as the sequential execution. Thread-level speculation is often used in hardware, or for limited/specific purposes, such as loop parallelization. Typical assumptions on the hardware support, available resources and concurrency level, potential value prediction, etc, often significantly differ from our setting, but there is some intersection to STM works, in particular in certain software-based TLS approaches, such as Galois [38]. We note that LiTM, the system we used for the deterministic STM baseline, includes experiments and comparisons with Galois, and reportedly outperforms it.

Multi-version data-structures. Multi-version data structures are designed to avoid write conflicts [12]. They have a history of applications in the STM context [15, 39], some of which utilize optimistic concurrency control [14]. 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 [14, 19, 41].

Blockchain execution. The connection between STM techniques and parallel smart contract execution was explored in the past [7, 9, 10, 20]. A *miner-replay* paradigm was explored in [20], 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 [7] introduced an *order-execute* paradigm (OXII) for deterministic parallelism. The ordering stage is similar to the schedule preparation in [20], 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 [9, 10] 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 [8] and several related works [43, 45] follow the execute-order-validate paradigm. As a result, the execution phase can abort unserializable transactions before ordering. Transactions in [17] 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. As noted, our Rust implementation is part of the Aptos code-base [1] and deployed in production. An independent Go implementation of Block-STM was recently adopted into the Polygon code-base [5].

A simplified, less technical, overview of the algorithm can be found in our blogpost [49].

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.]. BlockSTM implementation. https://github.com/danielxiangzl/Block-STM.

[4] [n.d.]. Diem codebase. https://github.com/diem/diem/tree/main.

[5] [n.d.]. Polygon codebase. https://github.com/maticnetwork/bor.

[6] 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.

[7] 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

[8] 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.

[9] 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.

[10] 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]

[11] Hagit Attiya and Jennifer Welch. 2004. *Distributed computing: fundamentals, simulations, and advanced topics*. Vol. 19. John Wiley & Sons.

[12] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[13] 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).

[14] Edward Bortnikov, Eshcar 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.

[15] Joao Cachopo and António Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. *Science of Computer Programming* 63, 2 (2006), 172–185.

[16] 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.

[17] Yang Chen, Zhongxin Guo, Runhuai 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.

[18] 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* (2019).

[19] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.

[20] 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.

[21] 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.

[22] 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.

[23] Alvaro Estebanez, Diego R Llanos, and Arturo Gonzalez-Escribano. 2016. A survey on thread-level speculation techniques. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–39.

[24] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.

[25] 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.

[26] 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* (2022).

[27] 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.

[28] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2006. *Stmbench7: a benchmark for software transactional memory*. Technical Report.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, portable and parameterless. *ACM SIGPLAN Notices* 49, 4 (2014), 499–512.

[39] 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.

[40] 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.

[41] Torvald Riegel, Pascal Felber, and Christof Fetzer. 2006. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*. Springer, 284–298.

[42] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 80–82.

[43] 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.

[44] 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.

[45] 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.

[46] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[47] 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.

[48] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional data structure libraries. *ACM SIGPLAN Notices* 51, 6 (2016), 682–696.

[49] The Aptos Team. [n.d.]. BlockStm: Aptos parallel execution engine. https://medium.com/aptoslabs/block-stm-how-we-execute-over-160k-transactions-per-second-on-the-aptos-blockchain-3b003657e4ba.

[50] The DiemBFT Team. 2021. State machine replication in the Diem Blockchain. https://developers.diem.com/docs/technical-papers/state-machine-replication-paper.

[51] 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*.

[52] 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.

[53] 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.

[54] 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.

[55] 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.