Access Specification Aware Software Transactional Memory Techniques for Efficient Execution of Blockchain Transactions

Supra Research

24 February 2025

Abstract

For a high-performance blockchain like Supra Layer 1, minimizing latencies across key components is crucial—such as data dissemination, consensus (or ordering), and transaction execution. While through significant innovations we have improved the first two, transaction execution remains an area for further optimization.

Software Transactional Memory (STM) is a widely used technique for parallel execution, with Aptos' BlockSTM pioneering its application for efficient blockchain transaction processing on multi-core validator nodes. PEVM [13] later adapted BlockSTM for EVM transaction execution. However, we identified a gap in existing STM techniques—while access specifications have been used in industry (e.g., Solana's user-provided read-write sets), they have not been leveraged to enhance STM efficiency.

Our experiments demonstrate that specification-aware STMs outperform their plain counterparts on both EVM and MoveVM. To maximize these benefits, we design specification aware SupraSTM (saSupraSTM), a novel algorithm that fully utilizes access specifications. Through extensive testing, we show that saSupraSTM outperforms both our specification-aware adaptation of Aptos' BlockSTM and specification-aware PEVM, setting a new benchmark for transaction execution efficiency in blockchain systems.

1 Introduction

Blockchains are transforming human society by enabling Byzantine fault-tolerant decentralized trust. Their potential is widely recognized as immense, but realizing this potential depends on designing efficient and effective systems. Supra's Layer 1 blockchain is built for extreme performance, offering vertically integrated services such as price feed oracles, verifiable randomness, automation, support for MultiVM (multiple smart contract platforms), blockchain interoperability solutions, and various innovations in Decentralized Finance (DeFi) applications.

To achieve this high level of integration, Supra Research rigorously examines every component and building block from both the security and performance perspectives. Having led innovations in consensus and data dissemination, we now focus on making significant advancements in efficient blockchain transaction execution. This led us to explore the existing literature and state-of-the-art systems designed for optimizing transaction execution on multi-core validator machines.

Aptos' BlockSTM [8] represents the current state-of-the-art, applying classical Software Transactional Memory (STM) [14] techniques to the execution of ordered blocks of transactions. STM techniques are typically speculative or optimistic, meaning that they attempt to execute as many transactions in parallel as possible while detecting and resolving conflicts. If a transaction reads a value that later changes due to another transaction, it must be re-executed to maintain correctness.

Supra has also made advancements in this domain, introducing SupraSTM [11], a novel schedulerless STM technique for executing blockchain transactions.

Naturally, the throughput of STM-based execution varies based on the level of transaction conflicts, aborts, and re-executions (re-validations), presenting both a challenge and an opportunity to optimize execution time further.

We observe the following:

- Solana leverages user-provided read-write sets as access specifications in its lock-profile-based iterative parallel execution strategy [17].

- Sui [7] also utilizes user-provided access specifications to execute transactions in a causally ordered manner.
- We further observe that it is possible to derive access specifications for public entry functions statically at the time of deployment of smart contracts using static data flow analysis. This is a one-time computation, enabling efficient utilization of these access specifications during transaction execution.

These access specifications essentially serve as conflict specifications for any parallel execution technique on an ordered blocks of transactions. This is where we see the challenge of improving STM as an opportunity and an avenue to optimize execution times further by leveraging access specifications. However, this paper does not focus on techniques to derive access specifications; rather, we investigate how to leverage them for efficient transaction execution. Specifically, we seek to answer the following research questions:

1. Do access specifications improve transaction execution throughput in existing STM techniques?
2. If so, what is the best STM technique to leverage these access specifications optimally?

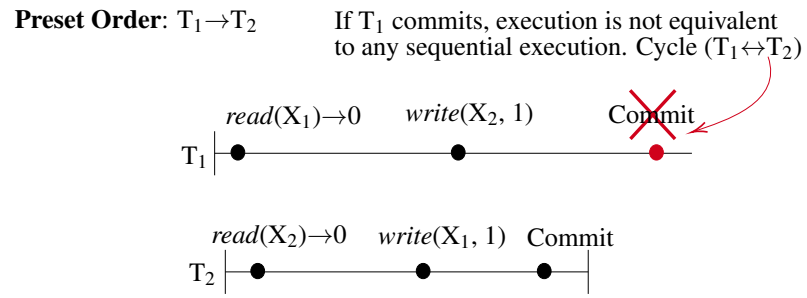
By affirmatively and constructively answering these questions, we aim to advance the frontiers of parallel transaction execution, driving significant improvements in blockchain scalability and efficiency.

The rest of the paper is organized as follows. Section 2 provides the motivating example for specification-aware parallel execution. In Section 3, we introduce the system model and provide a concise overview of the proposed algorithms. The experimental results and findings are presented in Section 4. Finally, we conclude the paper in Section 5.

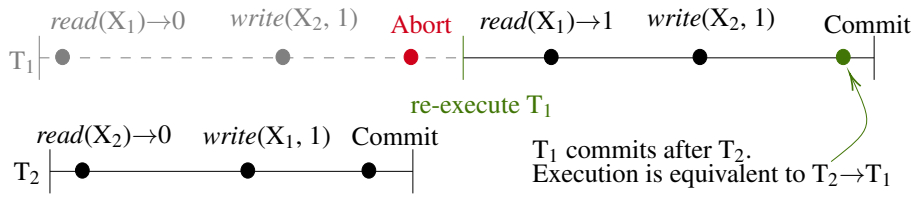
2 Overview of Specification aware STM

Consider the scenario shown in Figure 1, where we have two transactions T_1 and T_2 running concurrently. We consider that transaction T_1 must precede T_2 in the preset order (denoted $T_1 \rightarrow T_2$), without *a priori* knowledge of read-write conflicts in the set of accounts accessed, committing T_2 prior to the commit of T_1 may result in a safety violation. To illustrate this, consider the following execution: T_1 performs the read of an account X_1 (reading value 0), following which T_2 performs a read of an account X_2 (reading value 0), followed by a write of a new value 1 to account X_1. Observe that if T_2 commits at this point in the execution and if T_1 may wish to write a new value 1 to X_2 after the commit of T_2, then the resulting execution does not respect the preset order in any extension. Clearly, if transaction T_1 commits, the resulting execution is not equivalent to any sequential execution, as shown in Figure 1a. Alternatively, if T_1 aborts and then re-starts the execution, any read of the account X_2 will return the value 1 that is written by T_2, thus not respecting the preset order, as illustrated in Figure 1b. Consequently, the only possible mechanism to avoid this requires T_2 to wait until T_1 commits (shown in Figure 1c). Now consider a modification of this execution in which T_2 reads from X_3 and writes to the new account X_4. In this scenario, the transaction T_2 does not have a *read-from conflict* with T_1 allowing T_1 and T_2 to run in parallel with almost no synchronization between threads executing them. However, this is possible only if threads executing these transactions are *aware* of the conflict prior to the start of the transaction execution.

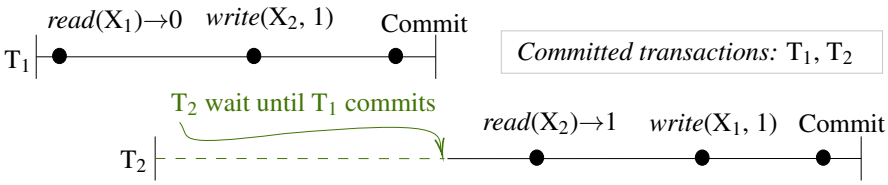
Consider Listing 2, which demonstrates the *transfer()* function of a token contract written in Solidity [16], a smart contract programming language for EVM-based blockchains. This function facilitates the transfer of tokens from the sender to the recipient, provided that the sender has a sufficient balance.



(a) Safety violation of preset order $T_1 \rightarrow T_2$



(b) Safety violation of preset order $T_1 \rightarrow T_2$



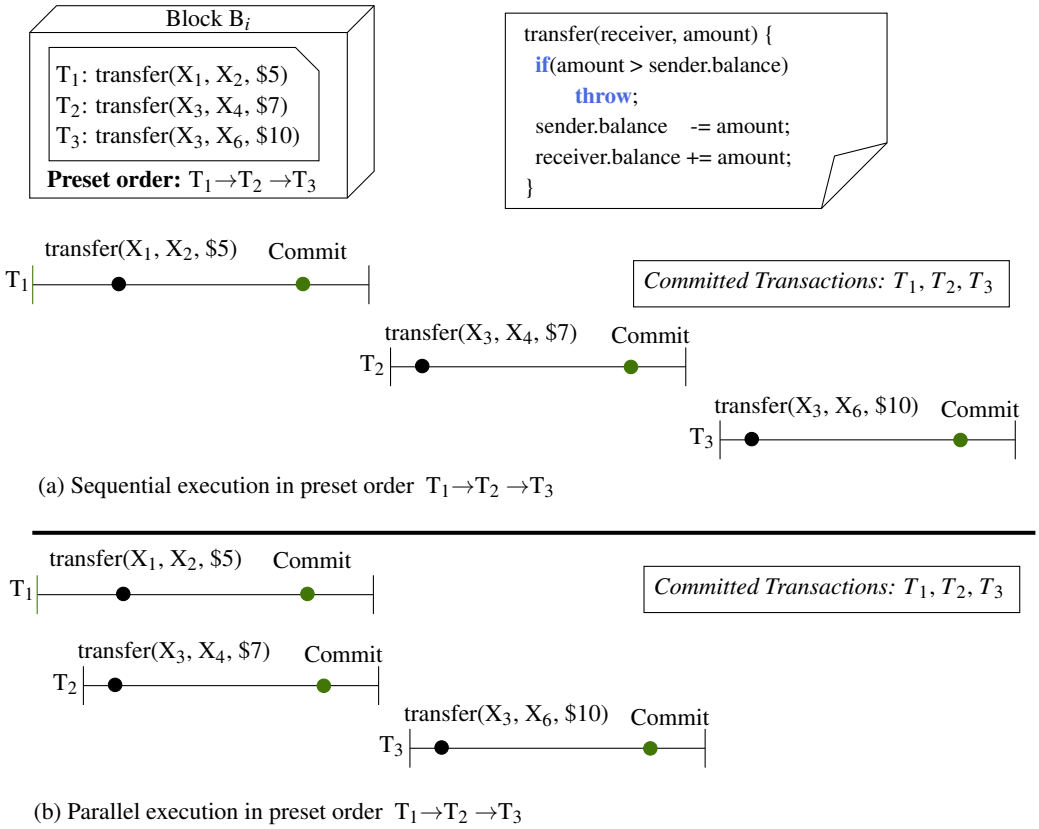
(c) Safe execution of preset order $T_1 \rightarrow T_2$

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

Listing 1: Transfer function

```
1  transfer(receiver, amount) {
2        if(amount > sender.balance)
3              throw;
4
5        sender.balance -= amount;
6        receiver.balance += amount;
7  }
```

Now, consider Figure 2 as a running example. As shown, we have a block B_i consisting of three transfer transactions T_1, T_2, and T_3, the preset order for these transactions is $T_1 \rightarrow T_2 \rightarrow T_3$. In Figure 2a, serial execution processes three transactions, T_1: transfer(X_1, X_2, \$5), T_2: transfer(X_3, X_4, \$7), and T_3: transfer(X_3, X_6, \$10), sequentially, despite T_1 and T_2 accessing different accounts, limiting throughput. In contrast, Figure 2b illustrates parallel execution, where T_1 and T_2 execute in parallel,



(a) Sequential execution in preset order $T_1 \rightarrow T_2 \rightarrow T_3$

(b) Parallel execution in preset order $T_1 \rightarrow T_2 \rightarrow T_3$

Figure 2 Efficient execution of transactions.

while T_3 waits for T_2 to commit, due to conflict of accessing X_3, this execution improves throughput.

An important observation here is that the accounts accessed by the transfer function are known in advance as input parameters to the transactions. This information can be used as a specification to prevent conflicts and avoid unnecessary transaction aborts in parallel execution.

In this paper, we explore the possibilities to maximize parallel execution throughput in this *read-write aware* model (as opposed to the *read-write oblivious* model [8]). We have built implementations of this approach for MoveVM [1] and Ethereum VM (EVM) based on the optimistic parallel execution strategies of BlockSTM. Our extensive experiment results demonstrate the advantages of our approach on several synthetic and real-world workloads. Additionally, we also demonstrate that this approach allows us to build workload *adaptive* parallel execution algorithms that can leverage a particular parallel execution algorithm depending on how conflicting a block of transactions is deemed to be.

3 Block Transactional Execution From Conflict Specifications

Model

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

$write_k(X, v)$, for a value $v \in V$, that returns *ok* or A_k, and $tryC_k$ that returns $C_k \notin V$ (*commit*) or A_k. The transaction T_k completes when any of its operations returns A_k or C_k.

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

Given a set of n transactions with a *preset* order $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$, we need a deterministic parallel execution protocol that efficiently executes block transactions using the serialization order and always leads to the same state even when executed sequentially. So, the objective is to parallelly execute the block transactions with the given preset order, such that parallel execution always produces a deterministic final state that is equivalent to the state produced by sequentially executing the transactions in the preset order.

We say that transactions T_i and T_j *conflict* if there exists a common account X in $Dset(T_i)$ and $Dset(T_j)$ such that X is contained in $Wset(T_i)$ or $Wset(T_j)$ or both. Furthermore, we say that T_i, T_j *read-from conflict* if $Wset(T_i) \cap Rset(T_j) \neq \emptyset$ and T_i appears before T_j in the preset order. Note that the definition of read-from conflict, unlike that of a conflict, relies on a preset order existing between the two transactions.

Algorithm overview

The core ideas behind our algorithm is that only independent transactions are executed in parallel, ensuring that no race conditions arise during execution. For any transaction T_k, if there exists a transaction $T_i \notin C\text{set}(k)$ that precedes T_k in the preset serialization order, the execution of T_k is deferred until T_i completes (here $C\text{set}(k)$ denotes the set of transactions that do not conflict with T_K). Additionally, after executing each transaction, validation is performed to ensure that if any specification for a transaction is incorrect and two dependent transactions are executed in parallel, the transaction higher in the preset serialization order can abort and re-execute. Thus, the output of the algorithm will be the same as that of the sequential execution.

The performance improvement over parallel execution techniques based on BlockSTM is achieved by reducing the number of aborts and re-executions. This is made possible by leveraging the knowledge of the transaction specifications: a transaction T_k is executed only after ensuring that all preceding transactions in the preset serialization order belong to its independence set, $C\text{set}(k)$. This targeted execution strategy minimizes conflicts and enhances throughput.

We implemented two different algorithms, specification aware SupraSTM and specification aware BlockSTM, based on how the scheduler utilizes the transaction independence (conflict) specification information for efficient parallel execution.

Specification Aware SupraSTM (saSupraSTM) This approach utilizes the conflict specifications of the transactions and the preset order to create a directed acyclic graph (DAG), which is used as a partial order and serves as input for the scheduler-less execution algorithm. In the DAG, transactions are represented as vertices, whereas conflicts among transactions are denoted as directed edges. The *indegree* field is added with each vertex (transaction) to track dependencies with prior transactions in the preset order; a transaction becomes eligible for execution when its indegree is zero. In $C\text{set}()$, if a transaction has its independence set to have all transactions prior to it in preset order, it contains no conflicts and will not have edges from any prior transactions, resulting in an indegree of zero. During execution, the non-zero indegree transactions wait for the

previous transactions to commit and remove dependencies. A transaction for which execution is completed decreases the indegree of all dependent transactions, allowing them to be executed. A custom parallel-queue executor is developed to manage transaction execution and allow efficient parallel execution through multiple threads.

Specification Aware BlockSTM (saBlockSTM) This approach optimizes the BlockSTM scheduler by integrating independence information (i.e., Cset()) associated with transactions, which is assumed to be provided as input. The scheduler is modified to allow transactions that are independent of all preceding transactions in the preset order to execute without validation. A greater number of such independent transactions enhances the execution efficiency. This method builds on top of the existing scheduler in PEVM [13] for EVM and BlockSTM [8] for MoveVM, incorporating modifications to effectively utilize set independence information. This approach works exactly as an optimistic speculative executor (the same as PEVM and BlockSTM) when the independence (conflict) information for all the transactions is unknown.

4 Experiments

In this section, we analyze the parallel execution latency and throughput of the saSupraSTM and saBlockSTM approach over the baseline sequential execution and state-of-the-art parallel execution approaches in both the EVM and MoveVM settings.

We ran our experiments on a single-socket AMD machine consisting of 32 cores, 64 vCPUs, 128 GB of RAM, and running Ubuntu 18.04 operating system with 100 GB of SSD.

Performance comparison:

- **EVM Experiments:** We compare saSupraSTM and saBlockSTM with Parallel-EVM (PEVM) [13], a version of BlockSTM [8] for EVM by the RISE chain [12]. The experiments are performed on REVM [3], an EVM written in Rust. We tested the performance on synthetic transactional workloads and historical workloads from Ethereum's Mainnet. In EVM, the gas fee is paid with every transaction to the Coinbase account, which belongs to the block proposer. In this design, each transaction updates the Coinbase account, causing a 100% conflict with all preceding transactions. In order to fix this, the fee is credited to the Coinbase account at the end of the block after being locally collected from each transaction.
- **MoveVM Experiments:** We compare the performance of saBlockSTM with BlockSTM [8]. The tests are performed on peer-to-peer synthetic workloads that are available with BlockSTM test-setup [2].

The experiments are carried out in an execution setting of in-memory using REVM and MoveVM, and do not account for the latency of accessing states from persistent storage. The throughput and latency numbers reported in the draft may change based on other factors in an end-to-end distributed execution setup, such as network latencies for block consensus, finality steps in the block processing pipeline, and markedalized state access from persistent storage during parallel execution. Additionally, note that preset serialization allows us to validate correctness by comparing the output state of parallel executions to that of the corresponding sequential execution.

The experiments were carried out 52 times, with the first 2 executions left as warm-up runs. Each data point in the plots represents average throughput (tps) and latency (ms), where each execution is repeated 50 times.

4.1 EVM Analysis

Figure 3 to Figure 6, show the relationship between varying threads or varying conflicts ($\alpha-$tail distribution factor) within the block, represented on the X-axis, on two important performance metrics:

throughput (tps), represented by histograms on the primary Y-axis (Y1), and *execution latency (ms)*, as line graphs on the secondary Y-axis (Y2).

Construction and Testing of Synthetic Workloads

The experiments are conducted on synthetic workloads to analyze throughput and latency under various execution conditions. A synthetic workload is a computational workload that does not depend on user input but instead simulates transaction types, concurrency patterns, or system loads. For example, an ERC20 transfer workload can be used to evaluate performance during high-volume transaction periods.

We analyzed Ethereum's Mainnet blocks to generate synthetic workloads and found that specific accounts (or smart contracts) are accessed predominately, indicating that real historical blocks follow a tail distribution of address accesses. Following these insights, we generated different synthetic workloads for our experiments, each designed to reflect varying transaction conflicts and parallelism, following tail distribution patterns. The distribution is based on the addresses accessed by transactions in the block. Additionally, for synthetic workload, we added another metric on the X-axis, the parameter α, which controls the heaviness of the tail in the Pareto distribution [10] and determines conflicts in subfigure (b). There are fewer conflicts in the block when α is set to smaller values.

- **ERC20 Workload (W_{erc20}):** This workload simulates multiple ERC20 token contracts, with each contract representing a distinct cluster. Transfer transactions occur between addresses within each cluster, which has its own ERC20 token. The number of transactions per cluster is determined using a Pareto distribution [10], while the sender and receiver addresses are selected uniformly

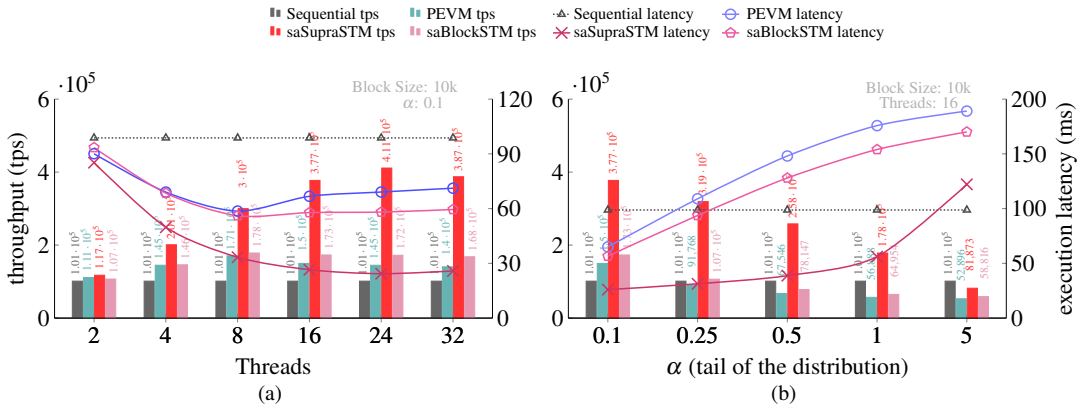


Figure 3 ERC20 transfer workload (W_{erc20})

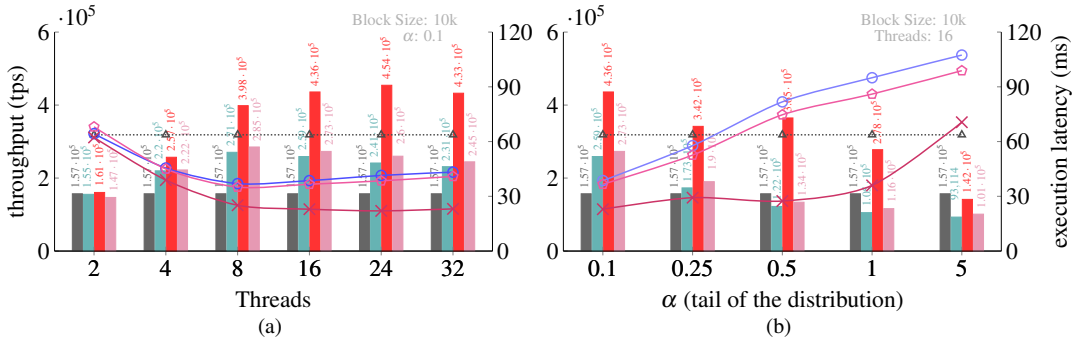


Figure 4 Mix workload (W_m)

within each cluster. This approach captures the distribution of transaction loads, resulting in certain clusters receiving higher transactions. The conflict specifications are derived using the sender, receiver, and contract addresses accessed by the transactions.

For testing, we generate 10k ERC20 token transfer transactions with 10k contract addresses (clusters), each cluster being mapped to an EOA address. Since each contract has a single EOA, transactions are effectively self-transfers. However, an EOA may initiate multiple transactions within a cluster, leading to conflicts, with a tail factor of $\alpha = 0.1$.

As shown in Figure 3a, increasing the number of threads improves performance. saSupraSTM achieves peak throughput at 24 threads with 411k tps, compared to 178k and 171k tps for saBlockSTM and PEVM, with average throughput of 299k, 157k and 143k tps, respectively. Notably, saSupraSTM achieves a 4× speedup over sequential execution and a 2.4× speedup over PEVM.

Latency trends show that optimistic approaches initially reduce execution time, but later increase due to higher abort rates. In contrast, saSupraSTM minimizes aborts through conflict independence, reducing both abort and (re-)validation costs. Similarly, in Figure 3b, with 16 threads, increasing block conflicts leads to a steady performance decline, more pronounced in optimistic approaches due to higher abort rates and increased waiting in saSupraSTM. At maximum conflicts ($\alpha = 5$), all approaches incur a higher overhead and perform worse than sequential execution.

- **Mixed Workload (W_m):** This workload combines 50% native ETH transfers and 50% ERC20 transfers, ensuring a balanced representation of both types of transactions. The workload is evenly divided between native ETH and token transfers. For native ETH transfer transactions, sender and receiver addresses are selected using a Pareto distribution [10].

  The number of ERC20 contracts is set to 25% of the block size, with workload generation similar to W_{erc20}. As a result, 2.5k contracts (clusters) comprise a block of 10k transactions.

  As shown in Figure 4, throughput and latency remain consistent with the W_{erc20} workload. However, due to 50% native ETH transfers (microtransactions) in the block, all approaches show a notable increase in throughput. The average tps is 356k for saSupraSTM, 238k for saBlockSTM, 229k for PEVM, and 157k for sequential execution. This translates to speedups of 2.27×, 1.52×, and 1.46× for saSupraSTM, saBlockSTM, and PEVM, respectively, compared to sequential execution. The maximum tps of saSupraSTM increased from 411k to 454k.

Deriving the conflict specifications

The conflict specifications are derived from the sender and receiver addresses for ETH transfer transactions, while for ERC20 transfers, they include the sender, receiver, and contract addresses. Note that in the current experiments, storage locations within the contracts are not considered in deriving these specifications. As a result, conflict specifications tend to be over-approximated.

Testing on Real-World Ethereum Transactions

We selected three distinct historical periods based on major events that may have impacted Ethereum's concurrency and network congestion. Each of these periods allows us to analyze performance under different historical workloads, providing insight into how major events, such as popular dApp launches and significant protocol upgrades, affect transaction throughput and network latency. It also helps us understand the limitations of parallel execution approaches under different network conditions, such as the gradual increase in conflicts in the block and the change in resource requirements over time.

We trace the accessed states of transactions within a historical block using the callTracer and prestateTracer [4], which provides a full view of the block's pre-state, the state required for the execution of current block. To derive the specification for transactions, the pre-state file is parsed to

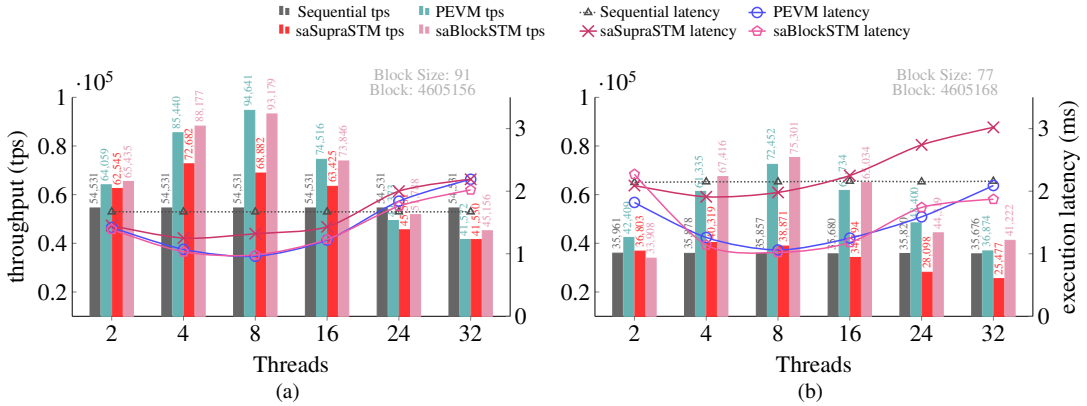


Figure 5 CryptoKitties historical period (W_{ck})

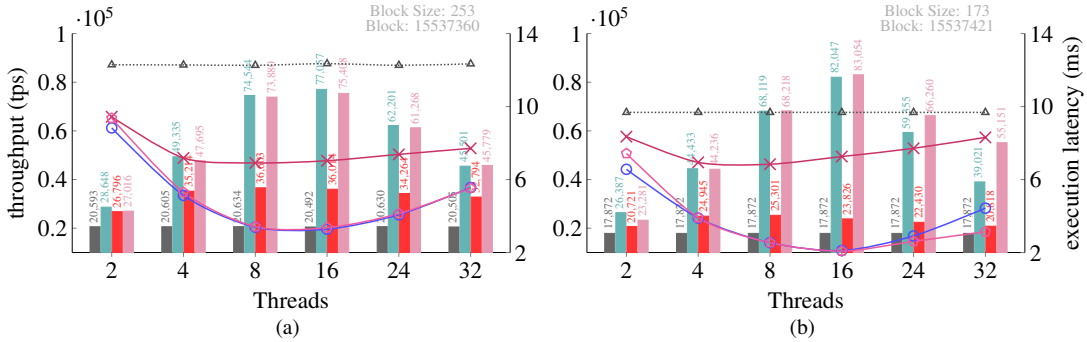


Figure 6 Ethereum 2.0 merge historical period (W_{e2})

identify all EOAs and contract addresses accessed in the block. The contract addresses and EOAs accessed by the transactions form the basis for constructing the conflict specification. Note that to derive a specification for a transaction T_i, all transactions (their pre-state accounts and contract addresses) before it in preset order are considered.

Note that real historical Ethereum blocks contain significantly fewer transactions compared to our synthetic tests. For our analysis, we selected larger blocks from different periods to ensure a representative evaluation. We analyze two blocks each from the CryptoKitties contract deployment and Ethereum 2.0 merge historical periods. In the detailed report, we will include a weighted average of 100 blocks before and after each historical event.

- **CryptoKitties Contract Deployment (W_{ck}):** The famous CryptoKitties [5] contract was deployed on block 4605167, after which an unexpected spike in transactions caused Ethereum to experience high congestion. We analyze block 4605156, which occurred before the contract deployment, and block 4605168, which took place after the deployment.

  Figure 5a demonstrates a significant speedup in block execution with parallel transaction processing, while Figure 5b highlights congestion in the post-contract deployment block. In saSupraSTM, conflicts are over-approximated due to contract-level and sender-address conflict specifications, as real blocks lack read-write access and contract storage access-level granularity. This leads to a performance drop compared to synthetic tests. With more precise conflict independence specifications and larger blocks, saSupraSTM could achieve superior performance, as observed in synthetic workloads.

  Throughput increases with additional threads, peaking at 8 threads, indicating the optimal point for speedup. Beyond this, threads compete for shared resources during transaction execution.

Notably, saBlockSTM outperforms in this workload both before and after contract deployment. The average tps for saBlockSTM, PEVM, and saSupraSTM drops from 69k, 68k, and 59k, before deployment to 54k, 53k, and 33k post-deployment. Additionally, despite smaller blocks in the post-deployment period, execution time increases for both sequential and other executors, highlighting the impact of congestion on overall performance.

- **Ethereum 2.0 Merge (W_{e2}):** The merge [6] took place in block number 15537393, which changed Ethereum's consensus to proof-of-stake along with other protocol-level changes and had an impact on transaction processing, block validation and network traffic in general. We analyze block 15537360 before the merge and 15537421 after the merge. Compared to the W_{ck} historical period, the Ethereum network has experienced increased block size, congestion, and conflicts due to increased user activity. As shown in Figure 6, the average throughput for saBlockSTM, PEVM, and saSupraSTM before the merge is 55k (2.68× speedup over sequential), 56k (2.73× speedup), and 33k (1.63× speedup), respectively. Post-merge, these values changed to 56k (3.17× speedup), 53k (2.98× speedup), and 23k (1.29× speedup). Notably, the increased speedup over sequential execution post-merge underscores the growing importance of parallel execution to improve throughput.

Observe that from one historical period to another (even with just these four blocks), despite the block size increasing, the overall throughput has decreased, highlighting the impact of congestion and resource requirements on overall performance. We will add a more detailed analysis on the large dataset in the detailed report.

In conclusion, our EVM analysis highlights the strong performance of saSupraSTM, demonstrating its potential to achieve near-theoretical maximum parallel execution through a conflict specification-aware design. However, in historical workloads, the lack of precise conflict specification, leading to excessive over-approximation, has allowed optimistic execution to perform better. With more accurate conflict specification in historical workloads, we can expect saSupraSTM to surpass optimistic execution, while saBlockSTM is expected to outperform BlockSTM. When no conflict specification is available, saBlockSTM is expected to perform exactly as PEVM.

4.2 MoveVM Analysis

Figure 7 describes our results for the implementation of saSupraSTM and saBlockSTM on the MoveVM when compared against the state-of-the-art BlockSTM and sequential execution. We perform these experiments on the BlockSTM benchmark test-setup [2], which runs a virtual machine for smart contracts in the Move language [1]. Performance (throughput as histogram on the Y1-axis and latency as line chart on the Y2-axis) is tested on peer-to-peer (p2p) transfer transactions, which essentially translate to contract transactions in the Move ecosystem.

In a block of 10k transactions, every p2p transaction randomly chooses two different accounts and transfers a fixed amount between them. The access specifications are derived by analyzing the sender and receiver of each transaction with all transactions preceding it in preset order. The conflicts in the block are determined by the number of accounts; specifically, when there are only two accounts, the load is inherently sequential (each transaction depends on the one prior to it). Performance is tested on varying threads (X-axis) on two different account setups: 100 (high contention, Figure 7a) and 10k (less contention, Figure 7b). We also tested performance on varying accounts (X-axis) with fixed threads (to 16 and 32), as shown in Figure 7c and Figure 7d.

As shown in Figure 7, the results reveal that saSupraSTM and saBlockSTM have throughput that is comparable to BlockSTM and are significantly better with 16 and 32 threads with 100 accounts for saSupraSTM in Figure 7a. As shown, saSupraSTM attains a maximum throughput of 54k at 16 threads, while saBlockSTM and BlockSTM attain a maximum throughput of 46k at 32 threads. The

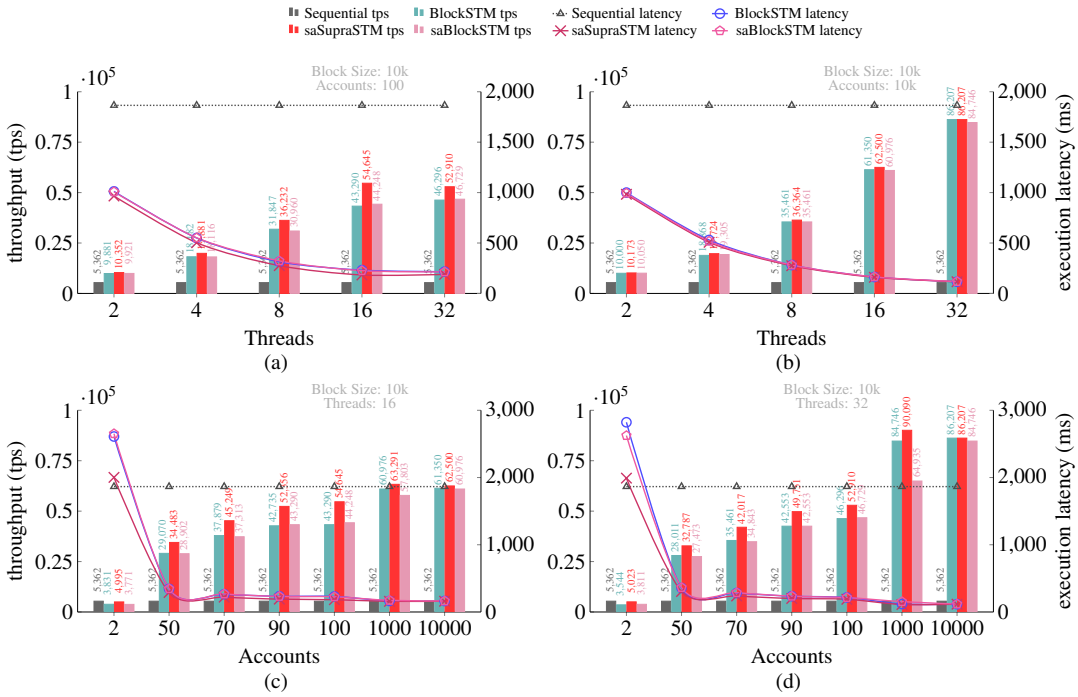


Figure 7 MoveVM p2p transfer workload (W_{p2p})

average tps is 34.8k, 29.99k and 29.89k for saSupraSTM, saBlockSTM and BlockSTM, respectively.

Similarly, in the other three experiments, the saSupraSTM outperforms other approaches, demonstrating the advantageous effects of knowing the conflict specification in advance. Note that with just two accounts for p2p transactions, it will be a 100% conflict case with a linear chain of conflicts, as shown in Figure 7c and Figure 7d. In this case, due to overhead, sequential execution outperforms parallel execution approaches. However, the tps increases linearly with the increasing number of accounts, until all transactions become independent, as can be seen that from 1k to 10k the tps is almost saturated for both 16 threads (Figure 7c) and 32 threads (Figure 7d). To improve on the worst case and to get more balanced performance in general (maximum throughput in all cases), we are developing a workload-adaptive parallel execution approach that is briefly discussed in Section 5.

Outcome 1 The experimental results from both EVM and MoveVM shows that saSupraSTM and saBlockSTM outperforms PEVM and BlockSTM. This highlights the significance of specification-aware parallel execution over optimistic execution.

Outcome 2 The experiments demonstrate that conflict specifications can be efficiently leveraged by saSupraSTM, enabling it to outperform saBlockSTM.

5 Discussion and Concluding Remarks

This report demonstrated the comprehensive benefits of conflict specification within smart contract execution, focusing on resource access patterns, dependency resolution, and parallel execution feasibility. Ongoing work also incorporates bytecode-level conflict detection, highlighting how specific parameters contribute to transaction conflicts.

Adaptive Implementation based on Conflict Threshold: The advantage of the conflict specification is that it allows us to determine how many pairwise conflicts exist in a block of transactions. We

demonstrate how we can leverage this for an adaptive implementation that can also handle high-conflict workloads. Specifically, by computing a *conflict threshold* for each block of transactions with minimal overhead, we deterministically fallback to sequential execution in case of high conflicts.

Our parallel execution techniques may allow for more localized dynamic *gas* fee marketplaces; if dependencies are specified a priori, transactions occurring in a congested contract of the blockchain state might be processed separately from others; to prevent a localized state hotspot from increasing fees for the whole blockchain network. For example, a popular *non fungible token (NFT)* mint could create a large number of transaction requests in a short period of time [9]. A blockchain that is based on a read-write aware setting can detect state hotspots upfront, such as the NFT minting to rate limit and charge a higher fee for transactions containing them [7, 15]. This enables ordinary transactions to execute promptly, while transactions related to the minting process are prioritised independently based on the total gas associated with them and resulting congestion.

References

1 S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. R. Rain, S. Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, page 1, 2019.

2 Block-stm implementation. `https://github.com/danielxiangzl/Block-STM`. [Online: accessed 15 January 2025].

3 BlueAlloy. Revm: A rust implementation of the evm. `https://github.com/bluealloy/revm`, 2023. [Online: accessed 15 January 2025].

4 Chainstack. Ethereum traceBlockByNumber API Reference. `https://docs.chainstack.com/reference/ethereum-traceblockbynumber`, 2025. [Online: accessed 15 January 2025].

5 CryptoKitties. CryptoKitties Website. `https://www.cryptokitties.co/`, 2024. [Online: accessed 15 January 2025, Contract Address: 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d, Creation Transaction: 0x691f348ef11e9ef95d540a2da2c5f38e36072619aa44db0827e1b8a276f120f4].

6 Ethereum Foundation. Ethereum 2.0 Merge. `https://ethereum.org/en/upgrades/merge/`, 2022. [Online: accessed 15 January 2025].

7 S. Foundation. All about parallelization. `https://blog.sui.io/parallelization-explained/`, January 2024. [Online: accessed 23 January 2025].

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

9 What is lazy minting? introducing a smarter yet economical way to mint nfts. `https://www.antiersolutions.com/what-is-lazy-minting-introducing-a-smarter-yet-economical-way-to-mint-nfts/`, January 2024. [Online: accessed 13 January 2025].

10 Pareto distribution. `https://en.wikipedia.org/wiki/Pareto_distribution`, 2023. [Online: accessed 15 January 2025].

11 S. Research. Block transactional memory: A complexity study. Technical report, `https://supra.com`, February 2025. `https://supra.com/documents/Supra-Block_Parallelization_Lower_Bound-Whitepaper.pdf`.

12 RISE Chain. RISE Chain Website. `https://www.riselabs.xyz/`, 2024. [Online: accessed 15 January 2025].

13 RISE Labs. PEVM: Parallel Ethereum Virtual Machine. `https://github.com/risechain/pevm`, 2023. [Online: accessed 15 January 2025].

14 N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

15 Solana fees, part 1. `https://www.umbraresearch.xyz/writings/solana-fees-part-1`, December 2023. [Online: accessed 10 December 2024].

16 Solidity Documentation. https://solidity.readthedocs.io/en/v0.5.3/. [Online: accessed 15 January 2025].

17 A. Yakovenko. Sealevel - parallel processing thousands of smart contracts. https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, September 2019. [Online: accessed 23 January 2025].