ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains

Haoran Lin*
Zhejiang University
Hangzhou, China
haoran_lin@zju.edu.cn

Hang Feng*
Zhejiang University
Hangzhou, China
h_feng@zju.edu.cn

Yajin Zhou†
Zhejiang University
Hangzhou, China
yajin_zhou@zju.edu.cn

Lei Wu
Zhejiang University
Hangzhou, China
lei_wu@zju.edu.cn

Abstract

Blockchain systems, especially EVM-compatible ones that serially execute transactions, face a significant limitation in throughput. One promising solution is concurrent transaction execution, which accelerates transaction processing and increases the overall throughput. However, existing concurrency control algorithms fail to obtain adequate speedups in high-contention blockchain workloads, primarily due to their transaction-level conflict resolution strategies.

This paper introduces a novel operation-level concurrency control algorithm tailored for blockchains. The crux of our approach is to ensure that only operations depending on conflicts are executed serially, while permitting concurrent execution of the remaining conflict-free operations. In contrast to conventional approaches that either block or abort an entire transaction upon detecting conflicts, our algorithm integrates a redo phase that identifies and re-executes conflicting operations. To facilitate this, we propose the SSA (static single-assignment) operation log, a mechanism to trace operation dependencies, thereby enabling precise conflict identification and efficient re-execution. Our prototype, ParallelEVM, is evaluated using real-world Ethereum blocks. Experimental results show that ParallelEVM achieves an average speedup of 4.28×, a marked improvement over the 2.49× speedup achieved by optimistic concurrency control.

*Both authors contributed equally to this work.
†Yajin Zhou is the corresponding author.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696063

CCS Concepts: • **Computing methodologies → Concurrent algorithms**.

Keywords: Concurrency, Operation-Level, EVM, Transaction Execution, SSA Operation Log

ACM Reference Format:
Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. 2025. ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3689031.3696063

1 Introduction

Following the growing prosperity of cryptocurrencies [34, 43], blockchain technologies have gained increasing attention. The introduction of smart contracts [13] has further expanded the applications of blockchains beyond cryptocurrencies, e.g., decentralized finance. But the limited throughput remains a significant impediment to the widespread adoption of blockchains.

To enhance blockchain throughput, efforts have primarily focused on reducing block time or increasing block size. Block time, the average time to generate a new block, is typically determined by the consensus protocol. Recent innovations have yielded various consensus protocols capable of generating blocks in a short time [30, 31, 40, 41]. Notably, Ethereum, a leading blockchain platform, has transitioned to the more efficient Proof-of-Stake (PoS) consensus [25]. Additionally, permissioned blockchains, such as Quorum, can adopt more aggressive consensus protocols to achieve shorter block times. Consequently, block time is no longer the primary bottleneck for throughput in these blockchains. However, the block size, determined by the number of transactions processed within the block time, has emerged as a significant challenge. A decrease in block time, without a corresponding improvement in transaction execution speed, inadvertently reduces block size, thereby undermining overall throughput. Therefore, our objective is to boost throughput by enhancing transaction execution speed.

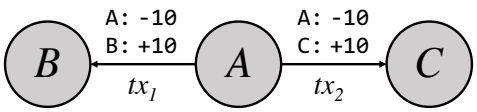


Figure 1. Token transfer example.

The Challenge of Concurrent Execution. Concurrent execution stands out as a promising approach to augment the performance of blockchain execution layers. By leveraging the parallelism inherent in modern hardware, this approach significantly improves the throughput of transaction execution. However, directly integrating conventional concurrency control algorithms into blockchains has proven to be suboptimal, largely attributed to the *hot spot problem* [2, 10, 24, 44]. For instance, in blockchains like Ethereum, a mere 0.1% of storage slots are accessed by 62% of storage operations, illustrating high data contention. Furthermore, the prevalent pattern for storage operations in blockchains is read-modify-write (RMW) [17]. When multiple concurrent RMW operations access the same storage slot, it results in intense data contention, significantly degrading the performance of conventional concurrency control algorithms.

The root cause behind the suboptimal performance of these traditional algorithms lies in their *transaction-level* conflict resolution strategies. They view transactions as atomic entities, processing them without insight into their internal operations. When faced with a conflict, these algorithms either block or abort the *entire* transaction, even if only a subset of operations are truly affected by the conflict. These coarse-grained parallelizing strategies prove inadequate for high-contention blockchain workloads.

An Insightful Example. To motivate a new conflict resolution strategy, we examine a representative token transfer scenario illustrated in Figure 1. The transaction tx_1 initiates a transfer of ten tokens from user A to B, while tx_2 initiates a similar transfer from user A to C. These transactions inherently conflict because they both access A's balance. However, the conflict on A's balance does not affect the updates to B's and C's balances. Their balances will increase by ten tokens irrespective of the conflict, assuming A has sufficient tokens to cover both tx_1 and tx_2. Thus, the RMW operations on B's and C's balances remain conflict-free, enabling potential concurrent execution. An efficient optimistic execution strategy might proceed as follows: (i) execute tx_1 and tx_2 concurrently akin to OCC, (ii) validate and commit tx_1, (iii) detect that tx_2 does not observe tx_1's update on A's balance, and (iv) re-execute the RMW operations on A's balance in tx_2 and then commit it. This paradigm addresses data conflicts at the *operation level*. Rather than blocking or aborting the entire transaction, only the operations dependent on the conflicting data are re-executed.

Proposed Approach. Based on the aforementioned insights, we propose a novel *operation-level* concurrency control algorithm designed to maximize parallelism, even under high-contention blockchain scenarios. Central to this algorithm is its ability to resolve data conflicts at the operation level. Specifically, only operations dependent on conflicting data are executed serially, while remaining operations proceed concurrently without blocking or aborting.

Our approach employs a variant of optimistic concurrency control (OCC). Consistent with conventional OCC practices, our approach initially runs transactions concurrently, recording accessed storage slots in a transaction-local memory. Subsequently, transactions undergo individual validation and commitment. In contrast to OCC, which aborts and restarts a transaction upon detecting conflicts, our method integrates a specialized *redo phase*. This phase identifies and re-executes all conflict-affected operations, ensuring that only these operations undergo re-execution, thereby mitigating performance degradation caused by data contention.

To identify and re-execute conflicting operations in the redo phase, we introduce the *SSA (static single assignment) operation log*, which not only records the inputs and outputs of EVM operations but also captures inter-operation dependencies. The SSA operation log ensures each variable to be assigned exactly once and defined before it is used. This approach establishes a definition-use chain for each variable, with the log explicitly stating the definition and all subsequent uses of the variable. When a conflicting variable is identified, definition-use chains are traversed to pinpoint all operations that depend on the conflicting variable, enabling precise and effective conflict resolution.

Our prototype system, ParallelEVM, is implemented based on Go Ethereum. Evaluations of this system reveal that ParallelEVM achieves an average speedup of 4.28× on real-world Ethereum workloads, a marked improvement over the 2.49× speedup of OCC.

Our Contribution. In summary, this paper makes the following contributions:

- **Novel Strategy.** We introduce an operation-level conflict resolution approach, enabling parallel transaction execution even in high-contention blockchain workloads.
- **New Techniques.** We present the SSA operation log, a new methodology for identifying and re-executing conflicting operations, accompanied by a comprehensive algorithm for its generation.
- **Efficient Prototype.** ParallelEVM demonstrates its adeptness in parallelizing transactions in real-world Ethereum environments. The evaluation indicates a tangible potential to augment the overall throughput by employing operation-level concurrency control algorithm.

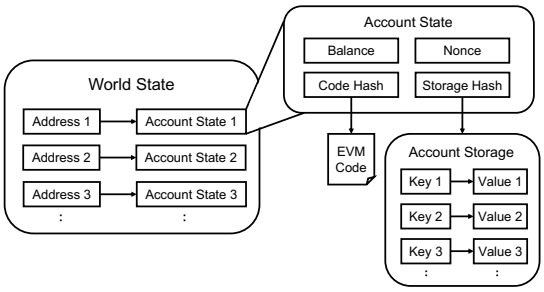


Figure 2. Ethereum world state.

2 Background

2.1 Blockchain

A blockchain is a distributed state machine maintained over a peer-to-peer network. It is organized as a sequence of blocks, each connected by cryptographic hash pointers. Within each block is a series of transactions that either transfer cryptocurrency or invoke smart contracts [43] to alter the blockchain's state. Typically, the design of a blockchain encompasses two distinct layers: a *consensus layer* ensuring all nodes, even those mutually distrustful of each other, agree on the same block history, and a *state layer* defining the state's structure and its transition rules.

Ethereum's State. Ethereum structures its state as the world state [43], which is illustrated in Figure 2. This state represents a relationship between account addresses and their respective account states. An Ethereum account has four components: (1) *balance*, the quantity of Ether, Ethereum's native cryptocurrency, owned by this account, (2) *nonce*, the number of transactions initiated by the account, (3) *code hash*, the hash pointer to the account's smart contract code, and (4) storage hash, the hash pointer to the root node of a Merkle Patricia tree [43], encoding the account's key-value storage (a mapping of 256-bit integers).

The rules governing Ethereum's world state transitions are stipulated by the Ethereum Virtual Machine (EVM) [43]. The EVM, a 256-bit word stack machine, comprises volatile byte-addressable memory and persistent key-value storage. The EVM code accesses the stack via instructions like `PUSH`, `POP`, `SWAP`, `DUP`, the memory via `MLOAD`, `MSTORE`, `MSTORE8`, and the storage via `SLOAD` and `SSTORE`. When executing a smart contract, the EVM runs the contract code and modifies the world state. Every EVM operational step has an associated "gas" cost, which the transaction's initiator covers.

Throughput Bottleneck. The throughput of a blockchain hinges on two parameters: *block time* and *block size*. Block time, typically fixed at the consensus layer, has historically been the throughput's constraining factor. For instance, the Proof-of-Work (PoW) consensus necessitates a prolonged block generation time (approximately 10 minutes per block for Bitcoin) to prevent simultaneous block creation. Consequently, Bitcoin [34] achieves a rate of only around seven transactions per second. However, several modern consensus protocols boasting rapid block generation have emerged and gained traction in both industrial and academic areas [19, 26, 30, 31, 40, 41]. Notably, Ethereum transitioned to the more efficient Proof-of-Stake (PoS) consensus [25], permitting shorter block times [1]. Besides, Quorum, a permissioned blockchain based on the Ethereum protocol, is reported to reach 2000 tps [3]. With such progress, the consensus layer is no longer the limiting factor for performance.

To augment throughput, increasing block size is a subsequent logical step. Yet, the size cannot be indiscriminately enlarged, as less efficient full nodes might lag behind the blockchain due to constraints in *storage space* and *execution speed*. Sharding presents a viable solution for the space constraint by partitioning the blockchain state into shards, enabling each node to manage only a fragment of the total state [11, 27, 32, 36, 42]. However, sharding does not rectify the speed issue, primarily attributed to the hot spot problem. Given that a majority of blockchain transactions invoke a few popular contracts, they are processed predominantly by a handful of active shards. This means that nodes in these shards handle a significant volume of transactions, restricting substantial block size growth. Thus, accelerating transaction execution is pivotal in achieving high-throughput blockchains.

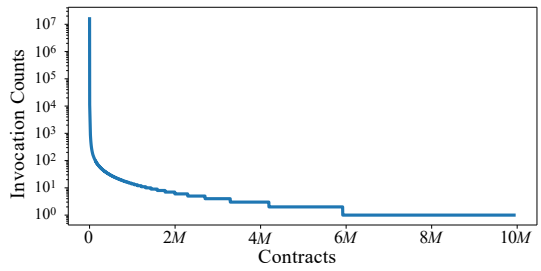
Limitations of Sequential Execution. The majority of blockchain systems process transactions within a block in a serial manner. While these serial execution models offer simplicity in reasoning, they do not capitalize on the inherent parallelism of contemporary commodity hardware, such as multi-core processors and SSDs. This underscores the need for introducing *concurrent transaction execution*.

2.2 Concurrency Control Algorithm

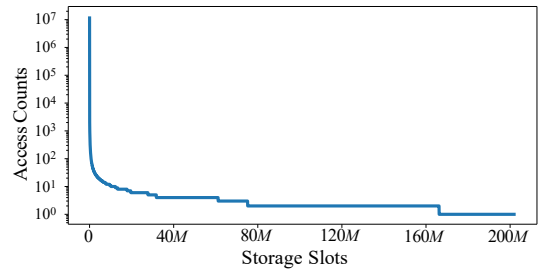
Concurrency Control in DBMSs. Traditional Database Management Systems (DBMSs) provide transactions with an illusion of isolation from concurrent transactions, ensuring they remain unaware of the effects of simultaneous transactions. Over time, a variety of concurrency control algorithms have been proposed to facilitate concurrent transaction execution while preserving this sense of isolation.

The essence of any concurrency control algorithm lies in its approach to *conflict resolution*. A conflict arises between two operations from distinct transactions when they access a shared object and at least one operation involves writing. Concurrency control algorithms can generally be categorized based on their conflict resolution methodologies: (1) *pessimistic algorithms* like 2PL[12], which employ locks to preclude conflicts, and (2) *optimistic algorithms* like OCC[29], which assume conflicts are rare and, when detected, abort and retry the transactions.

[1]For compatibility, the block time for Ethereum using PoS is still around 12 seconds at the current stage.



(a) Hot contracts.



(b) Hot storage slots.

Figure 3. The hot spot distributions for (a) contracts and (b) storage slots. The contract and slot indices (X-axes) are arranged in descending order based on their respective invocation and access counts. The counts for invocations and accesses (Y-axes) are depicted using a logarithmic scale.

```
1  contract ERC20 {
2    mapping(address => uint256) balances;
3    mapping(address => mapping(address => uint256))
         allowances;
4    function transferFrom(from, to, amount) {
5      _useAllowance(from, _msgSender(), amount);
6      _transfer(from, to, amount);
7    }
8    function _transfer(from, to, amount) {
9      require(balances[from] >= amount);
10     balances[from] -= amount;
11     balances[to] += amount;
12     ...
13   }
14   function _useAllowance(owner, spender, amount) {
15     require(allowances[owner][spender] >= amount);
16     allowances[owner][spender] -= amount;
17     ...
18   }
19 }
```

Figure 4. A fragment of an ERC20 contract in Solidity.

Concurrency Control in Blockchains. Blockchains impose a distinctive constraint on transaction execution, not present in conventional DBMSs: transactions must commit in the specific order outlined within the block. To address this challenge, researchers have adapted traditional concurrency control algorithms for blockchain contexts [2, 10, 17, 24]. In pessimistic approaches, such as 2PL variants, transactions acquire locks based on the sequence outlined in the block. Similarly, with optimistic methods like OCC variants, a transaction undergoes validation only after preceding transactions have been committed.

3 Motivation

3.1 The Hot Spot Problem

A defining feature of blockchain workloads is the *hot spot problem*. In blockchains, data accesses are highly skewed, where a minority of contracts or storage slots receive a disproportionate number of invocations or accesses. For instance, highly popular contracts like CryptoKitties [8] and certain crowdfunding agreements have notably strained the Ethereum network at times. To quantify this trend, we collected invocation and access counts for distinct contracts and storage slots on Ethereum between January 1, 2022, and July 1, 2022. Figure 3 reveals that merely 0.1% of the 10 million contracts are responsible for 76% of all invocations. Similarly, only 0.1% of the 200 million storage slots account for 62% of all storage access counts. Additionally, we observed that the ten most frequently invoked contracts represent approximately 25% of all contract invocations, with nine of them being ERC20 contracts [14] – the prevailing fungible token standard on Ethereum. These findings underscore the intense data contention present in blockchain workloads.

This hot spot problem considerably impairs the efficacy of traditional concurrency control algorithms. For those tailored to blockchain, pessimistic algorithms might see transactions enduring prolonged blocks or even being preempted by transactions of higher priority due to lock contention. In the case of optimistic algorithms, high data conflict rates result in numerous aborted transactions. Thus, the challenge of *formulating a concurrency control algorithm adept at resolving the hot spot dilemma in blockchain workloads* remains as an unresolved issue.

3.2 Toward Parallelizing an ERC20 Contract

An Example of Data Conflicts. Figure 4 depicts a segment of an ERC20 contract's implementation. This contract maintains its persistent state through two variables: the `balances` mapping, which denotes token owned by each account, and the `allowances` mapping, which monitors tokens authorized for third-party transfers. The `transferFrom` function allows message senders to transfer tokens on an owner's behalf. This function can fail if either the owner's token balance is insufficient (as seen in line 9) or the sender's allowances are inadequate (as seen in line 15).

Data conflicts typically emerge in ERC20 contracts when several transactions intend to distribute tokens from an identical sender address [17]. Consider the concurrent transactions:

$$tx_1 = \texttt{transferFrom}_D(A, B, value_1)$$
$$tx_2 = \texttt{transferFrom}_E(A, C, value_2)$$

,where function subscripts indicate the transaction's sender address. These transactions conflict because both target the same storage slot, `balances[A]`.

Transaction-level Conflict Handling Strategy. Traditional algorithms struggle to parallelize tx_1 and tx_2 due to the `balances[A]` conflict. However, a majority of operations within these transactions remain unaffected by this conflict (e.g., `balances[B]` and `balances[C]`). We denote such strategies as *transaction-level*, emphasizing that they treat a transaction as a whole, without regard to its individual operations. When faced with a conflict, these strategies either block or abort the complete transaction, leading to inefficiencies amid substantial data contention.

Operation-level Conflict Handling Strategy. Revisiting the previous example, under optimistic concurrency control, tx_2 fails validation because it does not observe tx_1's `balances[A]` update, resulting in an incorrect value. However, tx_2 does correctly update other slots. Using this observation, we enhance traditional OCC by incorporating a *redo phase* after the validation phase. Instead of aborting tx_2 after a failed validation, we re-execute line 10 using the updated `balances[A]` value. This allows for the resolution of the conflict through a single line of source code re-execution, enabling parallel execution of most operations within tx_1 and tx_2. This is termed as the *operation-level* strategy, focusing only on operations directly or indirectly impacted by conflicts. Given that most blockchain workload conflicts affect only a few operations [17], this approach can substantially mitigate performance issues linked to conflicts.

To ensure the correctness of the operation-level strategy, we introduce a concept termed *constraint guards*. Using a scenario from the earlier example, imagine if after executing tx_1, `balances[A]` does not have sufficient tokens for tx_2. Under these conditions, tx_2 must be aborted because of the balance verification in line 9. Our strategy should identify this violation and abort tx_2, rather than merely re-executing line 10. To facilitate this, we embed constraint guards into lines 9 and 15 that validate these essential conditions. During the redo phase for tx_2, if constraints are not met, the transaction is aborted.

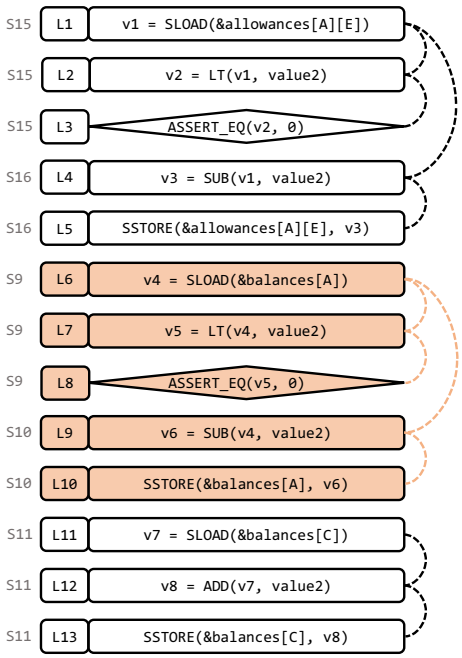


Figure 5. The SSA operation log of the transaction tx_2 in Section 3.2. Operations in diamond boxes (i.e., L3 and L8) are constraint guards. The corresponding line numbers in source code (Figure 4) are annotated on the left; the definition-use and use-definition chains are drawn on the right.

4 Challenges and Our Solution

4.1 Challenges

The cornerstone of our approach is to address transaction conflicts at the granularity of individual operations. While we have enhanced the conventional OCC with an additional redo phase that identifies and re-executes operations affected by conflicts, achieving this at the EVM bytecode level presents considerable challenges.

Identification of Conflicting Operations. For the redo phase to be both accurate and efficient, it's imperative to identify all operations contingent on conflicts, ensuring no under-estimation or over-estimation. Such conflicting operation identification requires precise dependency graph of all operations. Nevertheless, pinpointing these dependencies within the EVM bytecode is non-trivial. This is primarily because EVM operations don't manifest data dependencies explicitly. Instead of directly using the results of preceding operations as inputs, EVM operations obtain inputs from the *runtime context* – comprising the stack, memory, and storage. This layer of abstraction complicates the task of identifying the originating operations of these inputs.

Re-execution of Conflicting Operations. The successful execution of an EVM operation is contingent upon its specific runtime context. For instance, carrying out an `ADD` operation necessitates knowledge of the top two stack elements, while the `MLOAD` operation requires awareness of the memory state at that point. However, reconstructing the runtime context when re-executing conflicting operations only is challenging. Absent these contexts, the accurate re-execution of EVM operations becomes unfeasible.

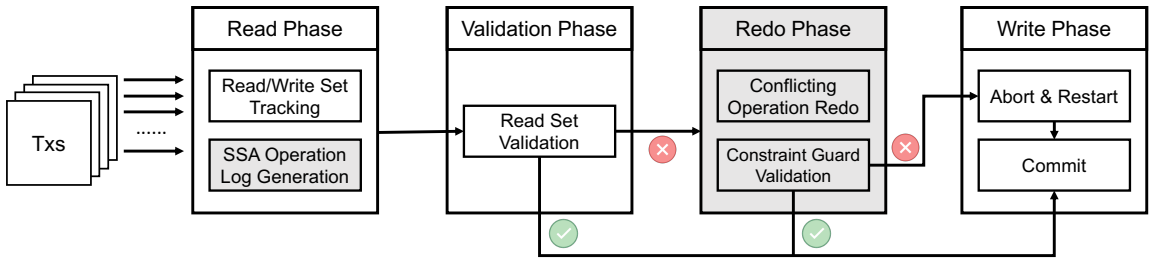


Figure 6. ParallelEVM workflow. The grey boxes are the central components for resolving conflicts at the operation-level.

4.2 Proposed Approach: The SSA Operation Log

To surmount the previously outlined challenges, we introduce a novel technique termed the *SSA operation log*. The foundational concept is that, instead of analyzing the EVM bytecode, we dynamically create the static single assignment form (SSA) [6] representation of the operations, which states the data dependencies explicitly. Concretely, we enforce that the inputs of each operation in SSA operation log must be explicitly delineated by : (i) immediate values, (ii) outputs of prior operations, and (iii) committed storage slots[2]. By adopting this approach, we bypass the need to access runtime contexts, thereby simplifying both the identification and re-execution of conflicting operations.

The SSA operation log offers clear use-definition and definition-use[3] relationships for operations. Throughout the redo phase, conflicting operations can be pinpointed by navigating the definition-use chains. Subsequent steps involve reconstructing operation inputs based on their definition operations and re-executing these conflicting operations. As an exemplar, Figure 5 delineates the SSA operation log for the transaction tx_2 discussed in Section 3.2. During the redo phase, the conflict associated with `balances[A]` can be traced back to the `L6` entry (`L6` is the first operation that introduces the conflicting value). By tracking the definition-use chains presented on the right of Figure 5, it becomes evident that operations spanning `L6` to `L10` are contingent upon the conflicting `balances[A]`. Therefore, we can replace `v4` in `L6` by the value committed by tx_1, re-execute instructions from `L7` to `L10`, and finally update `balances[A]` correctly.

5 ParallelEVM Design

5.1 Overview

Figure 6 illustrates that ParallelEVM employs a variant of optimistic concurrency control algorithms. However, ParallelEVM avoids aborting transactions immediately after the validation failure. It instead instigates a *redo phase* dedicated to conflict resolution at the operation level. Transaction execution in ParallelEVM is segmented into four distinct phases:

1. *Read phase:* ParallelEVM concurrently and speculatively executes transactions. For each transaction, ParallelEVM records all key-value pairs it accessed (read and write) and dynamically generates an *SSA operation log*, as discussed in Section 5.2.
2. *Validation phase:* Here, ParallelEVM validates the speculative executions from the read phase. Transactions progress to validation only if preceding ones are successfully committed. Within this phase, ParallelEVM revisits all key-value pairs from a transaction's read set, verifying consistency between initially read values and freshly retrieved ones. Success in validation propels a transaction to the write phase; failure diverts it to the redo phase.
3. *Redo phase:* At this phase, ParallelEVM attempts to resolve conflicts through re-execution of the conflicting operations. Leveraging the dependencies record in the SSA operation log, ParallelEVM identifies and re-executes all conflicting operations, as explained in Section 5.3. If the re-execution preserves all constraint guards, the transaction advances to the write phase for commitment. Failing that, the transaction aborts, necessitating a restart during the write phase.
4. *Write phase:* This final phase is where ParallelEVM consolidates transactions. If a transaction successfully pass either the validation or redo phase, ParallelEVM directly commits it, committing its write set to the storage. Conversely, if both phases fail, ParallelEVM aborts the transaction, necessitating a restart and eventual commitment.

5.2 SSA Operation Log Generation

This section illustrates the dynamic generation of the SSA operation log during the read phase. The log generation algorithm fundamentally seeks to trace data flows and pinpoint the defining operations of instruction inputs. An entry in this log contains:

- `LSN`: A unique log sequence number identifying the entry.
- `Opcode`: The operation's instruction code.
- `Operands`: Inputs of the operation.
- `Result`: The operation's output.

[2]A storage slot is termed 'committed' if it remains untouched by preceding operations in the ongoing transaction.

[3]A variable, v, situated on the left of an assignment statement, s, signifies a definition of v. Conversely, a variable, v, on the right of a statement, s, implies that a definition of v is utilized at s.

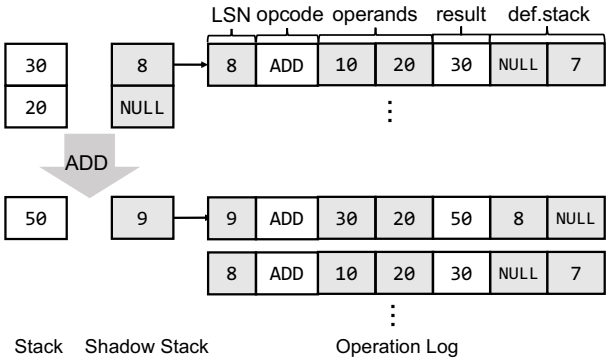


Figure 7. Stack tracking example.

- `Def`: Definition operations of input operands. This includes `stack`, `storage`, and `memory` sub-fields that trace data dependencies across the stack, storage, and memory. Further details on these structures and their generation rules are explored in Sections 5.2.1-5.2.3.

5.2.1 Stack Operation Log Generation. This section outlines the generation of SSA operation log entries for stack-only operations. For each operation, ParallelEVM records definition operations for all stack operands in the `def.stack` field. If a stack operand, `operands[i]`, is the `result` of a prior operation o_i, the `def.stack[i]` field is tagged with the `LSN` of o_i. Otherwise, it is assigned `NULL`.

ParallelEVM leverages a *shadow stack* to accurately populate the `def.stack` field, logging each stack item's definition operation. If an item is the `result` of a prior log entry, its shadow stack counterpart reflects the entry's `LSN`. Otherwise, it's marked `NULL`. Hence, ParallelEVM can retrieve definition operations for stack operands from the shadow stack, storing them in `def.stack`.

The shadow stack is managed akin to the actual stack. For the `PUSH` operation, which adds a constant to the stack, ParallelEVM pushes a `NULL` into the shadow stack. For `POP`, `SWAP`, or `DUP` operations, the shadow stack is adjusted as the actual stack. For other computational operations, shadow stack items (`lsns`) corresponding to stack operands are first identified and removed. If all `lsns` are `NULL`, indicating a constant result, a `NULL` is pushed into the shadow stack as the result is pushed to the actual stack. Otherwise, a new log entry is created, the `lsns` are stored in `def.stack`, and the entry's `LSN` is pushed to the shadow stack.

Figure 7 exemplifies the log entry generation for the `ADD` operation. Prior to the `ADD` operation, the stack holds two items: `30` and `20`. The shadow stack implies the first item (`30`) is the eighth log entry's result, and the second item (`20`) is a constant. Post-operation, ParallelEVM crafts a log entry for the `ADD` operation, pushes the result `50` onto the stack, and appends the new entry's `LSN` (i.e., `9`) to the shadow stack. Consequently, subsequent operations can realize that the top stack item (`50`) originates from the ninth log entry.

For storage and memory operations, the generation rules for `def.stack` fields are similar. But they should additionally maintain the `def.storage` and `def.memory` fields.

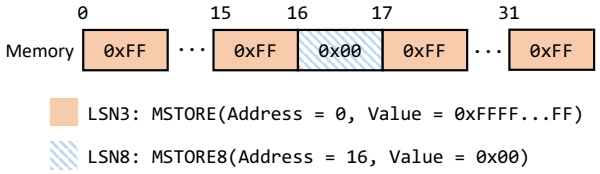
5.2.2 Storage Operation Log Generation. We now address the storage operations: `SLOAD` and `SSTORE`. `SLOAD` operations can be categorized into two types: (I) reading storage slots written by previous transactions (committed storage slots), and (II) reading values from prior `SSTORE` operations within the same transaction. The `def.storage` field is employed by ParallelEVM to differentiate these types. In the first scenario, the `def.storage` is set to `NULL` since these `SLOAD` operations are independent of any prior operations within the same transaction. In the latter scenario, it captures the `LSN` of the most recent corresponding `SSTORE` operation.

To monitor storage data flow, ParallelEVM utilizes two mapping structures during transaction execution. The first, `latest_writes`, records the `LSN` of the most recent `SSTORE` for each storage slot and is used to determine whether a slot has been written by the current transaction. Conversely, the `direct_reads`, which maps each storage slot to a set of `LSN`s, keeps track of the `LSN`s associated with all type I `SLOAD` operations. During the execution of an `SSTORE` writing storage slot A, ParallelEVM updates `latest_writes[A]` with the operation's `LSN`. For an `SLOAD` accessing slot A, if A is absent in `latest_writes`, it indicates that the slot has not been written by the current transaction and relies only on committed storage slots. In this case, the `def.storage` is set to `NULL`, and the entry's `LSN` is added to `direct_reads[A]`. Otherwise, `def.storage` is assigned the `latest_writes[A]` value, implying that the `SLOAD` reads the output of a prior `SSTORE`.

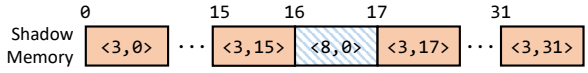
After the transaction execution, `direct_reads` precisely records all `SLOAD`s that read slots written by previous transactions (type I), assisting in identifying invalid slot reads during the redo phase (Section 5.3).

5.2.3 Memory Operation Log Generation. Within the EVM, memory operations, such as `MLOAD` and `MSTORE*`, are not uniformly aligned. This leads to potential overlap among multiple memory write operations, and individual memory reads might rely on several writes. An example showcasing this behavior with interleaved `MSTORE` and `MSTORE8` is presented in Figure 8a. Subsequent memory operations to this region must recognize their dependence on these two `MSTORE*` instructions.

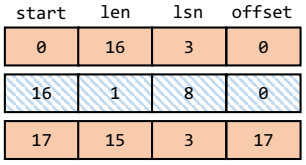
To track memory data flow, ParallelEVM incorporates a *shadow memory* strategy, analogous to the shadow stack mechanism. As delineated in Figure 8b, for every byte written by a `MSTORE*` operation, the corresponding byte in the shadow memory is marked with `<LSN, offset>`, where `LSN` is the write operation's log sequence number, and `offset` measures the distance from the current byte to the first one written by the `MSTORE*`.



(a) Interleaved MSTORE and MSTORE8.



(b) Shadow memory.



(c) The `def.memory` field corresponding to (b).

Figure 8. Memory tracking example.

Leveraging the shadow memory, ParallelEVM can record data dependencies of every memory read in the `def.memory` field. Precisely, the `def.memory` contains multiple `<start, len, lsn, offset>` tuples. These indicate that the bytes in the range `[start:start+len)` depend on the bytes in the range `[offset:offset+len)` from the `result` of the `lsn`-th log entry. For example, when a `MLOAD` operation reads a 32-byte memory span in Figure 8a, ParallelEVM populates the `def.memory` with the tuples depicted in Figure 8c.

5.2.4 Constraint Guard Generation. To ensure correctness, ParallelEVM imposes several constraints during the redo phase. These constraints are divided into control-flow, data-flow, and gas-flow categories. Control-flow constraints mandate that the re-execution in the redo phase replicates the path taken during the speculative execution in the read phase. For instance, the condition for a conditional jump (`JUMPI`) must remain consistent. Similarly, the destination for an unconditional jump (`JUMP`) should be invariant. When processing control-flow operations, ParallelEVM introduces constraint guards to ensure related operands (including jump conditions and destinations) remain unchanged. Specifically, if a shadow stack item corresponding to a control-flow operand is not `NULL` (indicating a non-constant operand), ParallelEVM appends an `ASSERT_EQ` log entry. This entry consists of an `operands` field representing the operand value and a `def.stack` field denoting the definition operation. During the redo phase, ParallelEVM compares the `operands` field with the `result` field of the log entry identified by `LSN` as `def.stack`. A match signifies constraint satisfaction, while a mismatch triggers a redo phase failure.

For data-flow constraints, ParallelEVM guarantees that the re-execution preserves the data interdependencies among operations. As an illustration, for an `MSTORE` operation, the target address must remain unchanged during the redo phase. If it's altered, ParallelEVM would not identify the operations dependent on this address-altered `MSTORE`. To safeguard this, whenever the target address of a runtime context operation (like `MLOAD` and `MSTORE`) is not constant, ParallelEVM adds an `ASSERT_EQ` log entry to guard the address operand.

Gas-flow constraints ensure that the transaction fee, determined by the total gas cost, remains valid after redo. Each EVM instruction consumes a specific amount of gas. EVM instructions are classified into constant cost instructions, which have a fixed gas cost, and dynamic cost instructions, where the gas cost varies depending on the execution context. For example, in the case of the `SSTORE` operation with a dynamic gas cost, changing a slot from zero to a non-zero value consumes more gas than performing the reverse operation. A successful redo requires that the total gas cost remains unchanged. To achieve this, ParallelEVM generates an `ASSERT_EQ` log entry for each dynamic cost instruction. These entries ensure that the gas cost of each re-executed operation matches the cost from the original execution.

5.2.5 Definition-Use Graph Generation. Building upon the `def` fields, ParallelEVM can identify all operations an individual operation hinges upon. However, the redo phase requires an inverse approach: identifying all operations depending on a given operation. To facilitate this, ParallelEVM crafts a definition-use graph, denoted as *DUG*, during SSA operation log generation. In *DUG*, each log entry translates into a node within the graph, and an edge from $entry_1$ to $entry_2$ implies that $entry_2$ utilizes $entry_1$'s result.

5.3 Redo Phase

Algorithm 1 describes the methodology for identifying and re-executing conflicting operations during the redo phase. Upon transaction validation failure, ParallelEVM obtains all conflicting storage slots and their correct values, as represented by the `conflicts` map in Algorithm 1. Then, ParallelEVM leverages the `direct_reads` map, as discussed in Section 5.2.2, and identifies all `SLOAD` log entries reading conflicting storage slots directly (see line 2). It subsequently amends their results to reflect the correct values (spanning lines 3-5). Using the depth-first search algorithm on the definition-use graph, ParallelEVM locates all conflicting operations depending on the prior conflicting `SLOAD` entries (see line 6). Following this, ParallelEVM sequentially re-executes these operations, excluding the prior `SLOAD` entries. In scenarios where an entry is a constraint guard, ParallelEVM verifies the satisfaction of the constraint (lines 9-11). Alternatively, it reconstructs the operation inputs based on results from definition operations specified in the `def` field (see line 13) and then re-execute the operation (line 14).

Algorithm 1: The algorithm for the redo phase.

```
Input: SSA operation log: oplog;
definition-use graph: DUG;
conflicting storage slot map: conflicts[keys]values.
Output: whether the redo phase is successful.
Function Redo(oplog, DUG, conflicts):
    sources ← SLOADs in oplog that read conflicts directly;
    for entry in sources do
        entry.result ← conflicts[entry.operands];
    end
    conflicting_ops ← DFS(DUG, sources);
    for entry in conflicting_ops / sources do
        if entry.opcode is ASSERT_EQ then
            expect ← entry.operands.value;
            current ← oplog[entry.operands.lsn].result;
            if current ≠ expect then return false;
        else
            reconstruct the inputs based on entry.def;
            re-execute entry and update entry.result;
        end
    end
    return true
end
```

5.4 Correctness

This section illustrates the correctness of ParallelEVM. We begin by establishing the equivalence between the SSA operation log and the EVM bytecode sequence.

Lemma 1. Given a transaction T_1, let its EVM bytecode sequence executed during the read phase be represented by $\{P[i]\}$, and let its corresponding SSA operation log be $\{L[i]\}$. If all entries in $\{L[i]\}$ are re-executed on storage S with all constraint guards satisfied, then the resulting outputs will match those of a direct re-execution of T_1 on the same storage.

Proof. Denote the re-executed EVM bytecode sequence of T_1 as $\{Q[i]\}$. In the initial execution (i.e., the read phase), each $L[i]$ maps uniquely to an EVM bytecode $P[j]$. We define the relationship between the index of SSA operation log (i) and the index of EVM bytecode (j) as $f(i) = j$, where f is a monotonically increasing function. The proof proceeds in two parts: (1) we posit that $\{P[i]\}$ and $\{Q[i]\}$ are identical and show that the re-execution of $\{L[i]\}$ and $\{Q[i]\}$ yield the same results; (2) we prove that $\{P[i]\}$ and $\{Q[i]\}$ are indeed the same sequence.

- **Part 1:** Assuming $\{P[i]\}$ and $\{Q[i]\}$ are identical, $\{L[i]\}$ and $\{Q[f(i)]\}$ would naturally be the same instructions. Given that $\{L[i]\}$ captures all storage-related instructions from $\{P[i]\}$ (and thus $\{Q[i]\}$), our goal is to demonstrate that every corresponding pair $L[i]$ and $Q[f(i)]$ yield identical results upon re-execution. This is done using induction. For the base case, $L[0]$ and $Q[f(0)]$ have identical results. For the induction step, presuming that for all $i < k$, $L[i]$ and $Q[f(i)]$ produce matching results, our next step is to verify this for $L[k]$ and $Q[f(k)]$. This is equivalent to show that the gas costs and inputs for $L[k]$ and $Q[f(k)]$ are the same. Gas-flow constraints guarantee that each instruction's gas cost remains unchanged in redo phase. Thus, $L[k]$ and $Q[f(k)]$ have the same gas cost. Now, we break down the inputs:
  - **Stack inputs:** Every EVM bytecode deterministically alters the stack structure, in both size and item order, regardless of its inputs. Given that $\{P[i]\}$ and $\{Q[i]\}$ are identical, they share the same stack structure, or equivalently, the same shadow stack. Hence, $L[k]$ sources its stack inputs from the results of the same operations as $Q[f(k)]$. Using induction, we deduce that the results of these definition operations align, ensuring the consistency of stack inputs between $L[k]$ and $Q[f(k)]$.
  - **Storage inputs:** When re-executing $\{L[i]\}$, adherence to data-flow constraints ensures that each $L[i]$'s target storage slot matches that of $P[f(i)]$. By induction, for $i < k$, each $L[i]$'s target storage slot aligns with that of $Q[f(i)]$. Furthermore, the target storage slots for $L[k]$ and $Q[f(k)]$, as stack inputs, are also identical. Hence, for $i <= k$, $L[i]$, $P[f(i)]$ and $Q[f(i)]$ uniformly access the same storage slot. That is, $L[k]$ and $Q[f(k)]$ share identical storage input dependencies, either reading from the same storage S or deriving from the results of identical operations, leading to the same storage inputs.
  - **Memory inputs:** Using a similar logic to storage inputs, we can establish that memory inputs for $L[k]$ and $Q[f(k)]$ are consistent.
- **Part 2:** To prove that $\{P[i]\}$ and $\{Q[i]\}$ are identical sequences, we employ a proof by contradiction. It's clear that $P[0]$ and $Q[0]$ are the same. Let's assume that $P[k+1]$ and $Q[k+1]$ are the first instruction that differ in these sequence. This implies that while $P[k]$ and $Q[k]$ are the same control-flow operation, they lead to distinct target addresses. Let's denote the counterpart of $P[k]$ in the SSA operation log as $L[f^{-1}(k)]$. Given that all control-flow constraints in $\{L[i]\}$ hold true, $L[f^{-1}(k)]$ must yield the same target address as $P[k]$. However, as part 1 demonstrated, the execution result of $L[f^{-1}(k)]$ matches that of $Q[k]$. By transitivity, $P[k]$ and $Q[k]$ must have congruent target addresses, which leads to a contradiction. □

Subsequently, we demonstrate that the partial re-execution of the SSA operation log in the redo phase is equivalent to a full re-execution.

Lemma 2. For a given SSA operation log $\{L[i]\}$, the subsequent two execution methods yield identical results: (1) executing $\{L[i]\}$ over storage $S_1 \cup S_2$; (2) first executing $\{L[i]\}$ over storage $S_1 \cup S_3$ and then partially re-executing entries dependent on S_3 by substituting S_3 with S_2.

Proof. Denote the executed entries in the first method as $\{L_1[i]\}$, and in the second method as $\{L_2[i]\}$. We prove that the results of $L_1[i]$ and $L_2[i]$ are identical using induction:
Base case: Both $L_1[0]$ and $L_2[0]$ are SLOADs targeted on the same address, leading to the same results due to the storage replacement.
Inductive step: Assume for all $i < k$, $L_1[i]$ and $L_2[i]$, the results of $L_1[i]$ and $L_2[i]$ are identical. We prove that this holds true for $L_1[k]$ and $L_2[k]$. Both $L_1[k]$ and $L_2[k]$ share the same input dependencies, either from prior operations (line 13 in Algorithm 1, these inputs are consistent due to inductive hypothesis that all prior operations produce the same results) or from storage (line 4 in Algorithm 1, these inputs are consistent due to the storage replacement). Thus, the results of $L_1[k]$ and $L_2[k]$ are identical. □

Finally, combining Lemma 1 and Lemma 2, we conclude the correctness of ParallelEVM.

Theorem 1. ParallelEVM yields results identical to those of serial transaction execution.

Proof. We establish the serializability of ParallelEVM by demonstrating its equivalency to OCC. From Lemma 2, we show that the partial SSA operation log re-execution during the read phase is the same as the full SSA operation log re-execution. Lemma 1 further establishes that a successful full SSA operation log re-execution is equivalent to the EVM bytecode re-execution. Notably, in ParallelEVM, even when the SSA operation log execution fails, it still results in the EVM bytecode being executed during the subsequent write phase. Hence, the mechanism in ParallelEVM – a redo phase followed by a write phase – has the same effect as the abort-and-restart write phase seen in traditional OCC models. Given that the serializability of OCC is well-established [29], we can confidently state that ParallelEVM produces the same results as the serial transaction execution. □

6 Implementation and Evaluation

We have implemented a prototype of ParallelEVM based on Go Ethereum v1.10.17, involving approximately 4200 lines of code changed.

6.1 Experimental Setup

Our evaluation was conducted on a machine equipped with an 8-core, 16-thread CPU and 16GB memory, running the Ubuntu 22.04 operating system, which mirrors the typical setup of an Ethereum node. Additionally, for comparison, we integrated optimistic concurrency control (OCC), two-phase locking (2PL), and Block-STM [18] into Go Ethereum. To collect historical transactions and states for evaluation, we deployed an Ethereum archive node as proposed by Feng et al. [16]. The workloads consist of Ethereum mainnet blocks ranging from block height 14,000,000 (January 2022) to 15,000,000 (June 2022).

Table 1. Speedups achieved by different algorithms.

Baseline	2PL	OCC	Block-STM	ParallelEVM
1×	1.26×	2.49×	2.82×	**4.28**×

6.2 Correctness Validation

To validate the correctness of ParallelEVM's implementation, we execute real-world Ethereum blocks using ParallelEVM and compare the resulted states with the Ethereum mainnet states. Ethereum maintains its state as a Merkle Patricia Trie [43], where every non-leaf node contains the cryptographic hash of its child nodes. Therefore, two Ethereum states are identical if and only if the root nodes of their respective MPTs match. We have run ParallelEVM to process the first 14 million blocks from the Ethereum mainnet, and ParallelEVM always produced a matching value of the MPT root for every block. This result demonstrates that ParallelEVM rigorously follows the rules defined in the Ethereum yellow paper [43].

6.3 Performance Analysis

Speedups in Real-World Ethereum. To measure the overall performance of ParallelEVM, we run ParallelEVM and other algorithms to process real-world Ethereum blocks. As shown in Table 1, ParallelEVM achieves an average speedup of 4.28× compared to Geth 1.10.17 (baseline). Figure 9 illustrates the detailed distribution of speedups: most blocks are accelerated by 2~7×. In a small subset of blocks (about 0.88%), ParallelEVM underperforms compared to the baseline, primarily due to time-consuming transactions that fail during the redo phase. In contrast, 2PL achieves a mere 1.26× speedup, as it tends to block and abort transactions in blockchain scenarios. Specifically, when transactions positioned earlier in the block attempt to obtain a lock held by later-sequenced transactions, the latter are aborted, resulting in significant performance degradation. OCC and Block-STM exhibit speedups of 2.49× and 2.82× respectively, which are inferior to ParallelEVM. This shortfall is attributed to the fact that both OCC and Block-STM need to re-execute the entire conflicting transactions, while ParallelEVM only re-executes the conflicting operations. It is worth noting that Block-STM's performance improvements in the original Block-STM paper are 20× in the Diem benchmarks and 17× in the Aptos benchmarks [18]. However, in our experiments, its performance gain is significantly lower. These performance differences can be attributed to the workload differences. Block-STM is originally evaluated using synthetically generated workloads, consisting primarily of randomly generated transfer transactions with varying conflict rates. These synthetic workloads fail to capture the hot spot features present in real-world blockchain systems. In contrast, our evaluation uses real Ethereum transactions. The hot spot data in real-world transactions limit the parallelism and

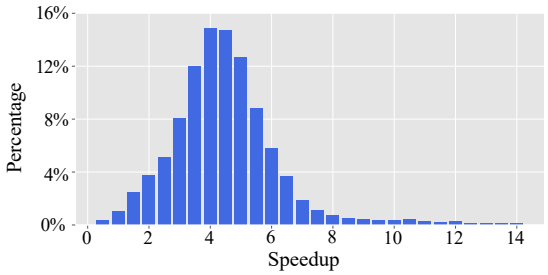


Figure 9. Speedup distribution for ParallelEVM.

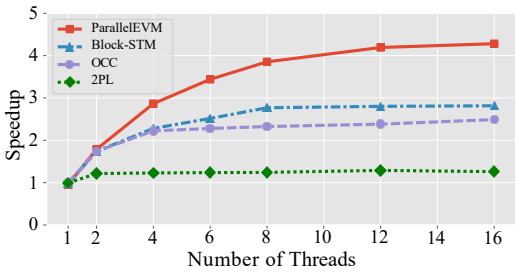


Figure 10. Impact of the number of threads.

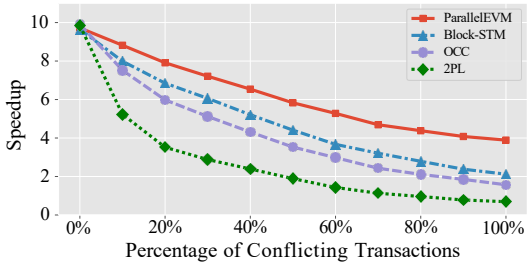


Figure 11. Impact of the conflicting transaction ratios.

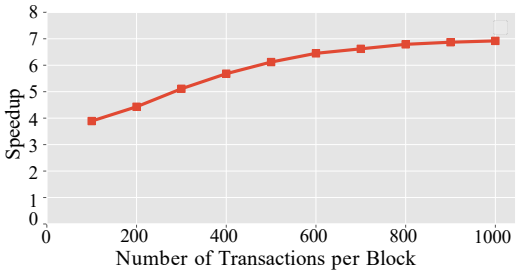


Figure 12. Impact of the block transaction number on ParallelEVM.

speedup achieved by Block-STM, as the critical path—defined as the longest transaction chain within a block that contains dependencies—tends to be much longer and becomes the bottleneck. Several works [17, 37, 38] also highlight this bottleneck of parallel systems in real-world blockchains, i.e., the optimal performance gain varies from 2× to 8×.

Impact of Thread Number. Figure 10 shows the speedups achieved by ParallelEVM and other algorithms at varying thread counts. The results demonstrate that ParallelEVM outperforms 2PL, OCC and Block-STM, indicating its superior scalability. This is primarily attributed to ParallelEVM's operation-level conflict handling strategy, which alleviates the performance degradation caused by transaction conflicts in environments with a high number of threads.

Impact of Contention. To demonstrate the effectiveness of ParallelEVM in various workloads, we simulate varying contention environments. This is achieved by packing ERC20 contracts into blocks while controlling the percentage of conflicting transactions. A scenario with a 0% ratio indicates a conflict-free environment, allowing all transactions to proceed concurrently without aborting or blocking. Conversely, at a 100% ratio, every transaction, except the first one, encounters conflicts.

Figure 11 illustrates the impact of conflict ratios. In low contention settings, ParallelEVM obtains similar speedups compared to OCC and Block-STM, attributed to the minimal overheads incurred during dynamic tracking and the generation of the SSA operation log (as discussed in Section 6.4). As contention intensifies, ParallelEVM begins to outperform other algorithms significantly. This is primarily because ParallelEVM is designed to re-execute only the conflicting operations when encountering conflicts, while OCC and Block-STM needs to abort and re-execute entire conflicting transactions.

Impact of Block Transaction Number. The number of transactions in a block also influences the performance of concurrency control algorithms. In today's Ethereum, the number of transactions within a block is fairly limited, often not exceeding 200. However, it's vital for ParallelEVM to be scalable and remain efficient even if future developments lead to larger block sizes. To this end, we simulate blocks with different sizes and evaluate the performance of ParallelEVM. Figure 12 illustrates the relationship between the number of transactions in a block and the subsequent speedup achieved by ParallelEVM. This figure shows a promising trend for ParallelEVM: as the block size enlarges with more transactions, ParallelEVM consistently exhibits higher speedup, showcasing its scalability and efficiency.

State Prefetching Optimization. Analyzing the runtime profiling data, it becomes evident that storage operations, especially `SLOAD`s, serve as the performance bottleneck for ParallelEVM. Accessing the persistent Ethereum state from the on-disk LevelDB database incurs notable latency. State prefetching is a promising technique to mitigate expensive

Table 2. Speedups achieved by different algorithms combined with prefetching.

Prefetch	2PL+	OCC+	Block-STM+	ParallelEVM+
2.89 ×	2.23×	3.25×	5.52×	**7.11**×

disk reads by caching storage slots in memory. To evaluate the performance of ParallelEVM with state prefetching techniques, we adopt a two-phase approach to block processing: an initial run dedicated to prefetching storage slots, followed by a performance-assessment run. We assess the speedups by measuring the duration of the second run. As depicted in Table 2, combining with prefetching, ParallelEVM achieves an average speedup of 7.11×. As comparisons, prefetching alone accelerates execution by 2.89×; 2PL, OCC, and Block-STM combined with prefetching achieve speedups of 2.23×, 3.25×, and 5.52× respectively. Although the actual speedups in real-world scenarios depends on the effectiveness of prefetching techniques, which is outside the scope of this paper, the above results suggest that ParallelEVM can cooperate better with prefetching techniques than traditional algorithms.

Pre-execution Optimization. Forerunner [5] introduced a speculative transaction execution algorithm for Ethereum. It utilizes the time window between when a transaction is known and when it is executed to predict and pre-execute transactions speculatively. Similarly, ParallelEVM can exploit this window to enhance its performance. By pre-executing transactions speculatively in a manner akin to the read phase, SSA operation logs can be pre-generated. During actual execution transactions that have been pre-executed can proceed directly to the validation phase, bypassing the read phase. Even if there are discrepancies in storage values retrieved during pre-execution, the redo phase can promptly reconcile the mismatch using the pre-computed SSA operation log. We simulate such optimization in ParallelEVM and observe an average speedup of 8.81×, underscoring ParallelEVM's capability for pre-execution.

6.4 Analysis of ParallelEVM Overhead

While ParallelEVM is designed to optimize high-contention workloads, it is crucial that it remains efficient and does not introduce undue overhead in scenarios absence of conflicts. We provide a comprehensive analysis of ParallelEVM's overhead in this section.

Overhead of SSA Operation Log Generation. The generation of the SSA operation log demands the maintenance of both the shadow stack and shadow memory in real-time. These are remarkably lightweight as the EVM operates as a stack machine interpreter and the main bottlenecks in transaction execution are storage operations. Experiments show that our SSA operation log generation algorithm incurs a modest average runtime performance overhead of approximately 4.5% per transaction. Furthermore, as represented in Figure 11, the overhead of ParallelEVM compared to OCC in conflict-free scenarios is negligible.

Overhead of Redo Phase. The overhead of redo phase is determined by the number of operations requiring re-execution. We substantiate that these operations are minimal. Primarily, our generation algorithm significantly reduces the size of SSA operation log. For the real-world Ethereum contract invocations, the average EVM instruction count stands at 2559, contrasted with the average SSA operation log length, which is 127. Consequently, the SSA operation log's size, comprises a mere 5.0% of the EVM instructions. This significant reduction can be attributed to our algorithm's adeptness in cutting down stack manipulation instructions like `PUSH` and `POP`, and instructions independent of storage slots. Hence, even in circumstances where every entry in the SSA operation log requires re-execution during the redo phase, it remains more efficient than the complete transaction re-execution in OCC. Moreover, experiments show that only seven log entries on average, equivalent to 0.3% of the original EVM instructions, typically undergo re-execution in the redo phase. The actual time spent on the redo phase is a mere 4.9% of the overall block processing time, with 87% of conflicting transactions successfully resolving conflicts during this phase. These results demonstrate that the overhead introduced by the redo phase is minimal.

Memory Overhead. On our experimental platform, the average memory consumption of ParallelEVM is 9.48 GB. In contrast, the official Geth consumes 9.08GB memory. Therefore, name incurs only 4.41% memory overhead, mainly due to the shadow stack and shadow memory.

7 Discussion

Scope. Although the implementation and evaluation are specific to Ethereum, our solution is also applicable to other blockchain systems. First, due to our modular design, ParallelEVM can be easily applied to other EVM-compatible blockchain systems. Additionally, our operation-level conflict resolution paradigm could be extended to other EVM-incompatible blockchains, particularly those with account-based state models and stack-based virtual machines. This is because our approach focuses on minimizing and efficiently handling conflicts at the operation level, a technique that is also applicable to other parallel systems [9]. Even though these blockchains may differ in their execution environments, the fundamental concepts of conflict minimization and resolution are universally applicable. Second, many blockchains (e.g., Polygon and BSC) closely follow Ethereum's development and share similar workloads, particularly due to the presence of hot spot applications. For instance, Uniswap is deployed on nearly every major EVM-compatible blockchain. As a result, these blockchains exhibit similar workload characteristics, a similarity that has also been reported by previous works [17, 37].

Limitations and Future Work. As illustrated in Figure 9, there are situations (<1%) where ParallelEVM might underperform serial execution. A potential solution is to bifurcate ParallelEVM into two phases: firstly, miner (or proposer) nodes would craft concurrent execution schedules, subsequently integrating these schedules into the blocks. Thereafter, validator nodes would execute block transactions adhering strictly to these predefined schedules. In this way, the miner (proposer) could formulate efficient operation-level schedules, ensuring consistent transaction execution acceleration on the validator side. Our future work is to generate operation-level schedules based on the SSA operation log.

8 Related Work

Transaction Decomposition. Transaction decomposition divides a transaction into multiple smaller sub-transactions. When a conflict arises within a sub-transaction, only that specific sub-transaction is aborted. Transaction chopping is a well-known transaction decomposition mechanism [39]. But it permits only a single sub-transaction to abort and thus produces coarse-grained decomposed transaction. MV3C [9] summarizes transactions in the form of a dependency graph and partially re-executes conflicting transaction when encountering conflicts. Faleiro et al. [15] designed piece-wise visibility (PWV) concurrency control protocol to make transactions' writes visible prior to the end of their execution. However, both MV3C and PWV require prior knowledge of transaction programs, either from manual annotation or static analysis, complicating their integration into existing blockchains. In contrast, ParallelEVM integrates the SSA operation log, enabling dynamic dependency graph generation without the need for prior knowledge.

Blockchain Execution Acceleration. Various studies have explored the integration of traditional concurrency control algorithms into blockchain systems. Saraph and Herlihy [38] introduced a concurrent execution engine that divides transaction processing into two phases: an initial concurrent phase where all transactions executed in parallel, discarding those that conflict; followed by a sequential phase, where the earlier conflicting transactions are processed sequentially. Unfortunately, their approach suffers performance degradation in high-contention workloads. Garamvölgyi et al. [17] proposed OCC-DA, a variant of OCC featuring deterministic aborts. While their methods enhance the determinism of OCC, they do not necessarily improve its performance. Zhang et al. [45] introduced BlockPilot, a parallel execution framework for blockchains. In BlockPilot, proposers employ a write snapshot isolation-based OCC algorithm (OCC-WSI) to execute transactions in parallel and generate block profiles that include transaction read and write sets. Validators then generate a transaction-level conflict-free schedule based on the block profile. In contrast, ParallelEVM offers a more granular approach to schedule transactions, without the need for introducing extra transaction information into blocks. Jin et al. [24] proposed a concurrency protocol emphasizing validator-side concurrency in permissioned blockchains. Their approach divide the transaction dependency graph, as generated by miners, into several sub-graphs to preserve parallelism and reduce communication costs. Nonetheless, their focus remains limited to transaction interdependencies. ParBlockchain [1] introduces an order-execute paradigm (OXII) for permissioned blockchains. It crafts a dependency graph for the transactions within a block, thus facilitating the parallel execution of non-conflicting transactions. OXII requires the prior knowledge of the read-write set via either static analysis or pre-execution; however, ParallelEVM has no such prerequisite. Moreover, the SSA operation log enables ParallelEVM to parallelly execute non-conflicting operations, moving beyond just transaction-level parallelism.

Software Transactional Memory (STM) represents an alternative approach for atomically executing transactions in parallel [7, 21–23, 33]. Recent work has explored the use of STM techniques for parallel smart contract execution. Dickerson et al. [10] introduced a "miner-replay" paradigm where miners determine a serializable concurrent schedule using an STM library and transmit this schedule to validators. Subsequently, the validators construct a "fork-join" program that deterministically replay the block in parallel. OptSmart [2] harness optimistic STM systems to construct transaction dependency graphs. Additionally, it maintains multi-versioned objects to mitigate write-write conflicts. Block-STM [18] combines OCC with a collaborative schedular that coordinates the execution and validation tasks among threads. The schedular leverages the write set of aborted transactions to determine future dependencies, ensuring aborted transactions are not restarted until the conflicts are resolved. Both ParallelEVM and Block-STM harness information from aborted transactions to optimize subsequent re-executions. However, ParallelEVM offers a more granular approach to conflict resolution. When a conflict arises, Block-STM blocks or restarts the entire transaction until the conflict is resolved. In contrast, ParallelEVM permits the execution of non-conflicting operations irrespective of any conflicts. Essentially, ParallelEVM offers an "out-of-order" execution: non-conflicting operations are initially executed, followed by the re-execution of conflicting operations.

Speculative execution is an alternative approach to accelerate transaction execution [4, 5, 20, 28, 35]. Forerunner [5] put forward a constraint-based approach for speculative execution on Ethereum. It speculates on multiple futures and accelerates transactions based on imperfect predictions whenever certain constraints are satisfied. In contrast to Forerunner, ParallelEVM can accelerate transaction execution even in situations where prior transaction knowledge is not feasible. Moreover, similar to Forerunner, ParallelEVM can utilize the transaction dissemination phase to pre-compute the SSA operation log speculatively. Consequently, during

the actual execution, these pre-computed transactions can bypass the read phase and resolve their mismatch reads in the redo phase.

9 Conclusion

In this study, we present ParallelEVM, an operation-level concurrent transaction execution system tailored for EVM-compatible blockchains. By identifying and re-executing conflicting operations, ParallelEVM ensures that conflict-free operations proceed concurrently without unnecessary blocks or aborts. Evaluations demonstrate that ParallelEVM significantly boosts transaction execution speed in real-world Ethereum scenarios.

Acknowledgments

We thank all anonymous reviewers and the shepherd for their invaluable comments and support. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant 62172360, U21A20464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

[1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. ParBlockchain: leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS '19)*. IEEE Computer Society, 1337–1347.

[2] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '19)*. 83–92.

[3] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. 2018. Performance evaluation of the quorum blockchain platform. arXiv:1809.03421

[4] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nickolai Zeldovich. 2020. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 1139–1154.

[5] 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 (SOSP '21)*. 570–587.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490.

[7] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. Association for Computing Machinery, 336–346.

[8] Dapper. 2017. CryptoKitties source code. https://gist.github.com/yogin/b88b105d9b2e332a5b59a3fd29cac962.

[9] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction repair for multi-version concurrency control. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, 235–250.

[10] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. Association for Computing Machinery, 303–312.

[11] T Elrond. 2019. A highly scalable public blockchain via adaptive state sharding and secure proof of stake.

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (1976), 624–633.

[13] Ethereum. 2022. Introduction to smart contracts. https://ethereum.org/en/smart-contracts. accessed: 2022-08-30.

[14] Vitalik Buterin Fabian Vogelsteller. 2015. EIP-20: token standard. https://eips.ethereum.org/EIPS/eip-20. accessed: 2022-08-30.

[15] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017), 613–624.

[16] Hang Feng, Yufeng Hu, Yinghan Kou, Runhuai Li, Jianfeng Zhu, Lei Wu, and Yajin Zhou. 2024. SlimArchive: a lightweight architecture for Ethereum archive nodes. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1257–1272.

[17] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. 2022. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery.

[18] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. 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)*. Association for Computing Machinery, 232–244.

[19] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, 51–68.

[20] Michael R. Head and Madhusudhan Govindaraju. 2009. Performance enhancement with speculative execution based parallelism for processing large-scale xml-based application data. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC '09)*. Association for Computing Machinery, 21–30.

[21] 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 (PPoPP '08)*. 207–216.

[22] 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 (PPoPP '08)*. Association for Computing Machinery, 207–216.

[23] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC '03)*. 92–101.

[24] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. 2021. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1.

[25] Sunny King and Scott Nadal. 2012. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August* 19, 1 (2012).

[26] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium*. USENIX Association, 279–296.

[27] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: a secure, scale-Out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. 583–598.

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems* 27, 4, Article 7 (2010).

[29] H. T. Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226.

[30] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security (FC '15)*. Springer, 528–547.

[31] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A decentralized blockchain with high throughput and fast confirmation. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, Article 35.

[32] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, 17–30.

[33] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, 166–176.

[34] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[35] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2005. Speculative execution in a distributed file system. *ACM SIGOPS Operating Systems Review* 39, 5 (2005), 191–205.

[36] George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical smart contract sharding with ownership and commutativity analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 1327–1341.

[37] Daniël Reijsbergen and Tien Tuan Anh Dinh. 2020. On exploiting transaction concurrency to speed up blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1044–1054.

[38] Vikram Saraph and Maurice Herlihy. 2019. An empirical study of speculative concurrency in Ethereum smart contracts. *CoRR* abs/1901.01376 (2019). arXiv:1901.01376

[39] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: algorithms and performance studies. *ACM Transactions on Database Systems* 20, 3 (1995), 325–363.

[40] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security (FC '15)*. Springer, 507–527.

[41] Yonatan Sompolinsky and Aviv Zohar. 2018. Phantom. *IACR Cryptology ePrint Archive, Report 2018/104* (2018).

[42] Jiaping Wang and Hao Wang. 2019. Monoxide: scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 95–112.

[43] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[44] An Zhang and Kunlong Zhang. 2018. Enabling concurrency on smart contracts using multiversion ordering. In *Web and Big Data*. Springer International Publishing, 425–439.

[45] Haowen Zhang, Jing Li, He Zhao, Tong Zhou, Nianzu Sheng, and Hengyu Pan. 2023. BlockPilot: A Proposer-Validator Parallel Execution Framework for Blockchain. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23)*. Association for Computing Machinery, 193–202.