2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-3986-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDCS57875.2023.00068

Smart Contract Parallel Execution with Fine-Grained State Accesses

Xiaodong Qi
Nanyang Technological University
xiaodong.qi@ntu.edu.sg

Jiao Jiao
Nanyang Technological University
jiao0023@ntu.edu.sg

Yi Li
Nanyang Technological University
yi_li@ntu.edu.sg

***Abstract*—As various optimizations being proposed recently, the performance of blockchains is no longer limited by the consensus protocols, successfully scaling to thousands of transactions per second. To further improve blockchains' throughput, exploiting the parallelism in smart contract executions becomes a clear solution to resolve the new performance bottleneck. The existing techniques perform concurrency control on smart contract transactions based on pre-determined *read/write sets*, which can hardly be calculated precisely. As a result, many parallelization opportunities are missed in order to maintain the correctness of transaction executions. In this paper, we propose a novel execution scheduling framework, DMVCC, to further increase the parallelism in smart contract executions, via more fine-grained control on state accesses. DMVCC improves over existing techniques with two key features: (1) *write versioning*, eliminating the write-write conflicts between transactions, and (2) *early-write visibility*, enabling other transactions to read the writes from a transaction earlier, before it being committed. We integrated DMVCC into the Ethereum Virtual Machine, to evaluate its performance in real-world blockchain environments. The experimental results show that DMVCC doubles the parallel speedup achievable to a 20× overall speedup, compared with the serial execution baseline, approaching the theoretical optimum.**

***Index Terms*—Parallel execution, smart contract, blockchain**

I. Introduction

The blockchain technology owes its initial popularity to Bitcoin [1]—a decentralized cryptocurrency without relying on any central authority. With the widespread utilization of *smart contracts* [2], the applications of blockchain have been extended to broader contexts, such as finance [3], supply chains [4], and healthcare [5]. Smart contracts are self-executing programs running on blockchain to govern interactions among mutually untrusted parties. Ethereum [2] is among the most popular blockchain platforms that support smart contracts, whose transactions take up around 70% of all the traffic.When a smart contract is deployed on the blockchain platform, its state (i.e., *data*) and execution logic (i.e., *functions*) are stored on all blockchain validators.

On the Ethereum, all miners pack multiple transactions into a block and add it to the existing chain after verification, following a *Proof-of-Work* (PoW) [1] consensus protocol. This process is often time-consuming and limits the throughput of the blockchain: for example, Ethereum only achieves a throughput of 30 transactions per second (TPS). Once a block is verified by the majority of the validators, all of them execute the transactions in the block and update the states of involved smart contracts. The transactions within a block are executed serially to ensure that state consistency is always maintained among all validators.

To improve the throughput of blockchain without increasing the block generation rate, one may choose to pack more transactions into the same block. Yet, larger blocks would naturally slow down the transaction execution at the side of the miners, in terms of a simple serial execution. An obvious way to speed up is to exploit the multi-core processors widely available and parallelize the transaction execution [6, 7]. Determining whether two transactions from the same block can be parallelized may not be trivial, especially when the transactions are generated from smart contract executions. On the other hand, as many new and more efficient consensus protocols being proposed recently [8, 9], the throughput bottleneck of blockchain systems is now shifting to smart contract executions. For example, both Conflux [8] and OHIE [9] are able to process simple payment transactions with a throughput of more than 5,000 TPS, several orders of magnitude faster than the original PoW consensus. Therefore, optimizations on smart contract execution start to have a major impact on the overall blockchain performance.

Correct parallel transaction execution has to meet the requirement of *deterministic serializability*, where the effect of the parallelization is equivalent to that of the serial execution in the order how transactions appear in every block. Two transactions may *conflict* with each other through their accesses to shared data (state). A simple idea to achieve deterministic serializability is to allow non-conflicting transactions to be executed in parallel and conflicting ones still to be executed serially. As pointed out in the previous works [10, 11], however, most blocks are bottlenecked on a single chain of conflicting transactions that need to be executed serially, dominating the overall execution time. Consequently, the overall speedup achieved by these approaches is capped at about 4× compared with serial execution. The main reason behind this is the high contention in smart contract transactions. For example, a shared counter of a popular contract is updated by almost all the transactions, leaving little opportunity for parallelization. Garamvölgyi et al. [10] proposed some practical coding paradigms, such as *conflict-aware token distribution* and *partitioned counters*, to reduce the contention of smart contracts at the application level. The downside of these techniques is that they may not be generalizable to all contracts, and we

2575-8411/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDCS57875.2023.00068

```
pragma solidity ^0.6.12;
contract Example {
  mapping(address => uint) public A;
  uint[] public B;

  function UpdateB(address x, uint y) public {
    uint idx = A[x];
    if(idx > 1) {
      for(uint i = idx; i > 1; i--) {
        B[i] = B[i-2] + y;
      }
    } else {
      B[0] = 0;
      assert(y <= 10);
      B[1] = B[1] + y;
    }
  }
}
```

Fig. 1: Example contract highlighting state access dependencies.

would like to address this issue by optimizing the low-level scheduler implementation to achieve maximum parallelism for transaction execution.

The key challenge for maximizing parallelism is to precisely compute transaction dependencies, i.e., to accurately infer state variables (read/write sets) accessed by every transaction. Existing solutions [6, 12] either do not infer the read/write sets automatically, or their coarse-grained static analysis [13] may miss opportunities for parallelization. For example, in Fig. 1, the state variable, "`B[i]`", depends on "`idx`", which further depends on the runtime value of "`A[x]`". But since the value of "`A[x]`" is only known at runtime, static analysis may over-approximate transaction dependencies. Some other parallel execution solutions [14, 15] adopt the *optimistic concurrency control* (OCC) strategy to avoid the computation of read/write sets. As for OCC, transactions are executed in parallel first without considering the dependencies, and then the ones, which violate the deterministic serializability, are aborted and re-executed until there is none to be aborted. The pitfall of these approaches is that a large number of transactions need to be re-executed when the contention between transactions is high.

In this paper, we address the challenge by analyzing smart contract code to determine the precise read/write sets of each program statement and enable more find-grained state accesses. Orthogonal to the program analysis technique, we also present a novel scheduling framework called *deterministic multi-version concurrency control* (DMVCC). DMVCC improves the parallelism of transaction execution in two key aspects: (1) it eliminates the write-write conflicts between transactions by preserving effects of all write operations as separate versions, which is referred to as *write versioning*; and (2) it allows transactions to read uncommitted writes through *early-write visibility* feature.

In particular, we propose a novel program representation for smart contracts, called *state access graph* (SAG), to record various information needed for determining the dependencies between transactions. An SAG is first lazily constructed based on the contract source code, containing partial information for certain state accesses. With the actual transaction data, the SAG will be completed dynamically with concrete runtime values and then used to calculate the precise data dependencies. For an operation depending on an unready state, DMVCC allows to retrieve the requested values from a most recent snapshot of global states, in order to determine transaction dependencies.

To eliminate write-write conflicts, the writes of different transactions to a same state are preserved as separate versions. During the execution, each transaction reads a proper version from these, which is written by the closest preceding transaction in the block. As a result, writes of different transactions to a same state do not conflict with each other, which further increases the parallelism. Many previous works [6, 16] only allow a transaction's writes to be read after its results are committed, to avoid causing cascading aborts. In contrast, DMVCC makes the writes of uncommitted transaction visible to other transactions under a more fine-grained control: it does so as soon as there is no abortable statement to be executed. This is known as *early-write visibility*, and thus, other transactions depending on those values can be executed much earlier.

Organizations. The rest of the paper is organized as follows. Section II describes the background and our motivation. Section III overviews the workflow of DMVCC and explains its key designs. Section IV gives details on DMVCC and shows the correctness proof. Section V provides implementation details and presents the evaluation results. Finally, Sections VI and VII compares DMVCC with the related works and concludes the paper, respectively.

II. Background

In this section, we provide necessary background and define concepts needed for the rest of the paper.

A. Blockchain and Smart Contracts

Blockchain. A *blockchain* is a shared and distributed ledger, which consists of a chain of *blocks*, maintained by a decentralized network of nodes. These nodes are either *light nodes*, *full nodes*, or *miners*. Light nodes only store block headers, and full nodes have to store and validate every block. A miner is a special full node that participates in mining to generate new blocks. We use the terms "full node" and "validator" interchangeably in this paper. Validators pack new transactions as blocks and append them at the end of chain, following some consensus protocols, such as *Proof-of-Work* (PoW) [1].

Smart Contracts. A *smart contract* is a self-enforcing computer program, which allows user-defined contractual rules to be programmed and executed automatically on blockchains. Ethereum [2] is the most popular blockchain platform that supports smart contracts. A smart contract on Ethereum is written in the Solidity language [17], which is compiled into a sequence of bytecode instructions, and then executed on the *Ethereum Virtual Machine* (EVM). An EVM is a stack-based machine with an instruction set. Data is stored in the persistent memory storage area, the contract-local memory (a contract obtains a fresh instance for each message call), or the stack.

States and Transactions. There are two types of accounts in Ethereum, namely, the *user accounts* and the *contract accounts*. A user account is associated with its Ether balance information, has no associated code, and can send transactions. A contract account has an associated executable code and its own storage as described above. The collective persistent memory storage of all accounts represents the state of a blockchain.

Definition 1 (Contract States). *Let C be the set of smart contracts deployed on a blockchain. Then $S = \{S_c \mid c \in C\}$ is the states of the blockchain, where S_c is the contract state represented by a set of key-value pairs that map 256-bit words to 256-bit words. The value of a contract state, termed as a* state item*, can be accessed with its key I, i.e., $S_c[I]$.*

A user may trigger the execution of the code associated with a contract account, by sending a transaction to the blockchain network, i.e., making a contract call. When a smart contract function is invoked through a transaction, the current contract state is retrieved from the blockchain, and the updated contract state is stored back when the execution finishes. Besides, there is another type of transactions, called Ether transactions, which merely transfer Ether between accounts without incurring any code execution on EVM. To ensure that transaction executions terminate, each computational step incurs a cost denominated in gas, paid by transaction senders. A sender specifies a maximum amount of gas it is willing to pay (called gas limit), and if the charge exceeds the limit, the execution is terminated and rolled back, and the sender is not refunded.

Blocks and Snapshots. In a blockchain, a *block* B_l contains a sequence of transactions $\langle T_1, \ldots, T_m\rangle$ and a block header. Once a block is appended successfully, validators execute the transactions in the block, following the order specified in B_l, to update the blockchain states. Since all transactions are stored persistently on the blockchain, we may easily recover the states of blockchain at a certain block height. We use S^l to denote the l-th state snapshot, which is the blockchain state after executing all the transactions up to the l-th block. The set of all state snapshots, denoted as $\{S^0, S^1, \ldots\}$, is referred to as the *StateDB* of the blockchain.

B. Smart Contract Parallel Execution

The serial execution of traditional blockchains limits their throughput significantly. A direct solution is to leverage the multiple cores available to execute multiple transactions in parallel, which is well-studied in databases [18, 19]. These protocols commonly ensure *serializability*, where the effect of concurrent execution is equivalent to a serial execution in *some* order. The order, however, may vary for different executions, thus validators, running concurrent execution independently, may enter inconsistent states. The parallel executions in blockchain should additionally meet the *deterministic serializability criteria*, as defined in Definition 2, which promises that all validators obtain the same result for every block.

Definition 2 (Deterministic Serializability). *A schedule for a batch of transactions $\langle T_1, \ldots, T_n\rangle$, is* deterministically serializable *if its effect is equivalent to that of the serialized execution, which conforms to the transactions' commitment order, $\langle T_1, \ldots, T_n\rangle$.*

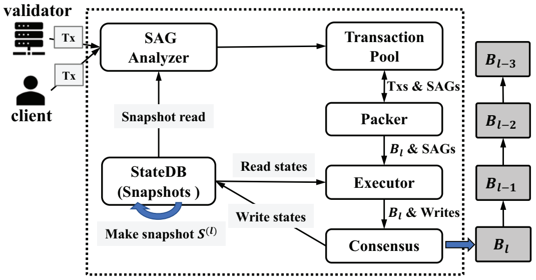


Fig. 2: The workflow of DMVCC on a single validator.

Many recent works [6, 7, 14, 20] explore the design space of parallel transaction execution for smart contracts. On one hand, some of them assume that the accurate read/write sets of transactions are readily available, which poses various practical challenges. For example, FISCO BCOS [12] requires users to specify the read/write sets explicitly to support parallelization of transactions. Such a setting is not applicable to smart contracts. On the other hand, some works [10, 21] employ the *Optimistic Concurrency Control* (OCC) strategy to execute transactions in parallel without read/write sets. With OCC, all transactions read state items from a state snapshot to drive the executions without reading writes of other transactions. As a result, all transactions can be executed in parallel. After the parallel execution, validators abort and re-execute the transactions that violate deterministic serializability.

However, according to the reported results [10], the speed-up achieved by existing approaches is far from linear on real-world Ethereum workload. This is mainly due to the lack of inherent parallelism on the real-world workloads—many frequently accessed shared variables force transactions to be executed sequentially, on a few critical paths. These approaches perform coarse-grained transaction-level concurrency controls without considering the logic of smart contracts, thus they cannot exploit the potential parallelism by analyzing the state access patterns at the statement level. In this paper, we seek to develop an alternative approach that adapts to the existing Ethereum architecture, to achieve much better parallelism by reducing conflicts between transactions.

III. Overview

In this section, we overview the high-level workflow of DMVCC and demonstrate it with an example.

A. Workflow

DMVCC is a novel scheduling approach to maximize the parallelism in smart contract transaction execution. To achieve this goal, DMVCC enables more fine-grained state accesses, where executions are synchronized at the statement level, as opposed to the transaction level. Such a fine-grained execution control is achieved by a precise analysis of the contract code.

Fig. 2 presents a generic workflow of DMVCC on a single validator, and the same workflow is replicated on all validators.

When receiving a transaction from a client or other validators,[1] each validator first analyzes the code of the invoked contract to infer the state items (read/write sets) to be accessed during the execution. The analysis results are captured by a statically-constructed *state access graph* (SAG), which is a custom data structure to record necessary information for DMVCC to produce the execution schedule (cf. Section III-B). The *SAG analyzer* may also retrieve some contract state values from the latest snapshot S^{l-1}, to further refine the SAG. Then, the processed transactions are stored in the *transaction pool*, along with their SAGs, waiting to be scheduled for executions later. Next, the *packer* periodically selects a number of transactions from the transaction pool to form a new block B_l, same as the original Ethereum does. The *executor* of each validator executes this block of transactions in parallel, following the schedule produced by DMVCC and updates the global state S when necessary. Finally, block B_l is appended to the current ledger after a consensus phase, and the state writes are applied on the *StateDB* to produce a new snapshot S^l.

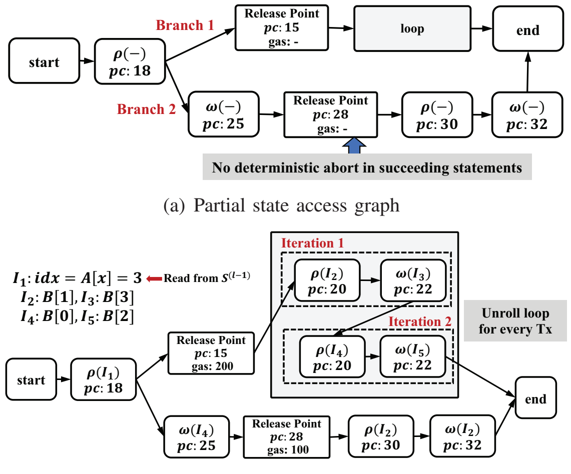
When a new valid block is mined by other validator successfully, the current validator attempts to retrieve the corresponding SAGs of the block cached in the local transaction pool. Due to the delay of transaction transmission, the transaction pools maintained by different validators may not be synchronized. If a transaction in the block is missing from the local pool, the validator constructs a SAG for it on-the-fly. Surely, the validator can also execute it without any information of the read/write set as what OCC does. Our approach still promises the correctness of parallel execution even if SAGs of some transactions are missing. Finally, the transactions and their corresponding SAGs are sent to the executor for parallel execution.

B. DMVCC by Example

Here, we illustrate the construction of SAGs and the generation of execution schedule with an example.

1) State Access Graph and Its Construction: A *state access graph* (SAG) is a simplified *control-flow graph* (CFG) [13], from which the nodes performing no read/write operation are removed. Particularly, we first build a *partial state access graph* (P-SAG) statically from contract source code and then use concrete values from a specific transaction to refine the P-SAG and generate a *complete state access graph* (C-SAG). Fig. 3 exemplifies the construction of P-SAG and C-SAG based on a transaction that calls the smart contract in Fig. 1.

A P-SAG, as depicted in Fig. 3(a), has a unique *start* and *end* node, and the other nodes in between are either (1) read or write nodes, (2) loop nodes, or (3) release points. A node with $\rho(-)$ or $\omega(-)$ corresponds to a read or write operation, respectively. Since the P-SAG is built statically, there may not be enough information to determine which state items will be accessed during the execution, without the actual parameters of function calls. We use a placeholder "$-$" in place of any unresolved access. Loops are represented as *loop nodes*, if they cannot be solved statically. Then these loops will be unrolled when transaction data is provided. The *program counters* (PCs), included in SAG nodes, maps operations to the corresponding EVM instructions.



(a) Partial state access graph

(b) Complete state access graph.

Fig. 3: The state access graph of the smart contract in Fig. 1.

Besides, some nodes are labeled as *release points*, which indicate that there exists no abortable statement beyond these points that may cause an abort. When executions reach release points, values of relevant state items can be made visible earlier to other transactions, instead of waiting till the end of the execution. For example, other transactions can read the written value for I_4 safely, without any risk of an abort, once the execution of Branch 2 reaches the release point. Additionally, to ensure there is enough gas to finish the execution, a *gas* field is included in each release point, which gives an upper bound estimation to the gas needed for the remaining statements.

When a new transaction arrive, a validator refines the P-SAG for the smart contract invoked into a C-SAG with concrete input of the transaction, as is shown in Fig. 3(b). The input parameters of the transaction and state snapshots are used to resolve the keys used in state accesses, which may not be known statically. For instance, in Fig. 1, the key used in a state access at Line 10 is "`idx`", which depends on the runtime value of "`A[x]`" (Line 7).[2] With the absence of "`idx`" value, a naïve solution would be to treat the entire array "`B`" as being accessed by the transaction exclusively, blocking simultaneous accesses from other transactions, which is over pessimistic and reduces opportunities for parallelization. In contrast, DMVCC reads the value of "`A[x]`" from a latest committed snapshot S^l to infer the value of "`idx`", without blocking accesses to the rest of the array. This helps enable more fine-grained state access controls and improves the parallelism. Suppose the snapshot value of "`idx`" is "3". Then the for-loop at Line 9 is also unrolled twice, which expands the loop node in the P-SAG into multiple nodes in the C-SAG. Then, a gas estimation tool is used to fill the *gas* fields in release points.

[1] When receiving a transaction sent by a client, a validator relays it to others through the Ethereum P2P network.

[2] For the sake of simplicity, variable names are used as keys here. In practice, a unique encoding is used to avoid name collisions with other smart contracts.

If the analysis is inaccurate, i.e, the state values used to infer keys are overlapped by other transactions, DMVCC will abort and re-execute the transaction to guarantee the deterministic serializability. More details about the construction of SAGs are presented in Section IV-A.

2) Transaction Conflicts and Access Sequences: The goal of DMVCC is to generate schedules that execute non-conflicting transactions in parallel, where *transaction conflicts* are defined formally as follows.

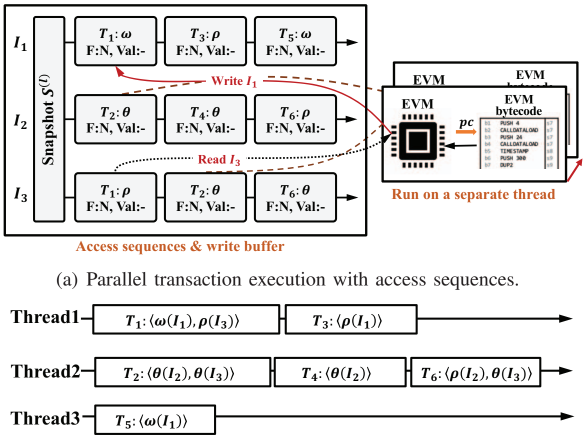
Definition 3 (Transaction Conflicts)**.** *Given two transactions $T_i, T_j \in B_l$ and their C-SAGs, AG_i and AG_j, T_i conflicts with T_j if there is a state item I that meets one of following conditions: (1) $\rho(I) \in AG_i \wedge \omega(I) \in AG_j$; or (2) $\omega(I) \in AG_i \wedge \rho(I) \in AG_j$. Otherwise, T_i and T_j are non-conflicting.*

Note that two transactions writing to a common state item are non-conflicting in our case. This is because write-write conflicts are eliminated by DMVCC's *write versioning* feature. At the high-level, all versions of writes performed by different transactions are stored in an *access sequence*, which can be used to resolve the correct values for read operations (see Section IV-D for details).

Definition 4 (Access Sequences)**.** *Let $\langle T_1, \ldots, T_n \rangle$ be a sequence of transactions in B_l. For a state item I, its access sequence w.r.t. B_l is denoted as $L_I = \langle T_{p_1} : \alpha_{p_1}, \ldots, T_{p_k} : \alpha_{p_k} \rangle$, such that $p_i \in \{1, \ldots, n\}$, $\alpha_{p_i} \in \{\rho, \omega, \theta\}$, and for any $i < j$, $p_i < p_j$ holds. The symbols ρ, ω, and θ indicate that the corresponding transaction T_{p_i} performs a read, a write, or both read and write on I, respectively. M_l denotes the set of access sequences for all the state items to be accessed by the transactions in B_l.*

An access sequence records the access types, the values to be read/written, and status of the access operations for a state item, in the order in which these operations appear in a block B_l. For example, the left hand side of Fig. 4(a) shows the access sequences of three state items, I_1, I_2, and I_3, to be accessed by the transactions from a block B_l. In each rectangle, the field "F" indicates the operation's status, and "Val" stores the value read or written by the transaction. They are initialized as "N" (not finished) and "$-$" (empty), respectively.

According to this sequence, a read operation of a transaction can be resolved to an appropriate version of the state item. If there is no write prior to a read, e.g., "$T_1 : \rho(I_3)$", the transaction T_1 should read the value of I_3 from the latest snapshot S^{l-1}. Otherwise, the read should be blocked until the latest prior write is finished. While the transactions are being executed, the validator inserts values of new writes to the access sequences and allows the blocked transactions to read them. If all the state items to be read by a transaction are ready, it then can be scheduled for execution. Meanwhile, an access sequence serves as a buffer, which holds intermediate versions of a state item for every operation. By storing all versions of writes, reads do not have to block later writes; write-write conflicts are also avoided, e.g., both "$T_1 : \omega$" and "$T_5 : \omega$" on I_1 can be scheduled to run in parallel.



(a) Parallel transaction execution with access sequences.

Thread1
$T_1: \langle \omega(I_1), \rho(I_3) \rangle$
$T_3: \langle \rho(I_1) \rangle$
Thread2
$T_2: \langle \theta(I_2), \theta(I_3) \rangle$
$T_4: \langle \theta(I_2) \rangle$
$T_6: \langle \rho(I_2), \theta(I_3) \rangle$
Thread3
$T_5: \langle \omega(I_1) \rangle$

(b) Execution scheduling of transactions over three threads.

Fig. 4: Example of parallel transaction execution based on DMVCC protocol.

3) Execution Schedule Generation: In the transaction execution stage, a validator executes the transactions in a block to obtain the results. Fig. 4 demonstrates the parallel execution of the six transactions, $\langle T_1, \ldots, T_6 \rangle$, of the block B_l. During the execution, an EVM instance reads state values from the access sequences and writes updates back, with the fields "F" and "Val" propagated accordingly. For example, the EVM instance executing T_1 first updates the "Val" field of the write operation on I_1 (i.e., "$T_1 : \omega$" at the top left), and sets the "F" field to "true" (finished). This essentially creates an intermediate version for I_1, and the transaction T_3 is allowed to read from this version. A transaction is ready to be scheduled for an execution, when the state items it depends on have all been written to. At this moment, the validator will create a new EVM instance to execute this transaction. If there are multiple transactions executed in parallel, these EVM instances perform the read/write access sequences concurrently and our design promises a safe synchronization.

Fig. 4(b) presents a schedule for the six transactions on three threads. At the very beginning, T_1, T_2, and T_5 are not blocked by any other transactions, hence they are executed in parallel on the three threads. They may directly read values from the latest snapshot. T_3 is scheduled on Thread 1, after T_1 has written the value of I_1 to the access sequence (L_{I_1}). Similarly, T_4 has to wait for T_2 to finish. This schedule synchronizes at the transaction level, and it is still far from ideal: Thread 3 stays idle after executing T_5, due to the inherent dependencies between transactions. In Section IV, we show that the parallelism of the transactions can be improved by adopting early write visibility and commutative write, which enables a more fine-grained control over state accesses.

IV. Protocol Design

In this section, we first introduce the construction of state access graphs as well as access sequences (Section IV-A). Then, we present algorithms for the schedule generation (Section IV-B) and early write visibility along with some optimizations (Section IV-C). Finally, we discuss the correctness of DMVCC (Section IV-F).

A. Preprocessing

Since the structure of a P-SAG resembles that of a CFG, we may reuse the skeleton of a CFG and remove nodes other than read and write operations. The main challenge in refining a P-SAG into a C-SAG is to resolve the keys of state items to be accessed during the execution, for a particular transaction. The keys used to perform a state item access may depend on other state values, inputs of transactions, and global parameters, such as the block heights and timestamps (we treat them as special transaction inputs), which may not be determined statically. For an access key I, its *state access dependency* is denoted as $\mathcal{D}_I(V, E)$, here V is the set of state values and E is the set of inputs to a transaction. Specifically, $V = \{Val(I_1), \dots, Val(I_i)\}$ and $E = \{e_1, \dots, e_j\}$.

We obtain set V from the latest snapshot S^{l-1} and the set E from a given transaction. Concrete values of the dependencies are used to execute the contract code (only a forward slice of the contract code is executed in our implementation), and the relevant fields of the C-SAG nodes are updated dynamically. In particular, the state access keys and the state items are resolved with their actual values. If the state item values read from the snapshot become stale, making the resulting C-SAG inaccurate, the abort mechanism still promises a recovery and ensures correct execution results. Next, after the C-SAGs are completed for transactions, the validator constructs the access sequences for all the state items to be accessed. This is done by a simple traversal of the C-SAGs.

B. Schedule Generation

Once the access sequences of the state items to be accessed by the transactions in B_l are ready, a validator starts to execute them concurrently, as shown in Algorithm 1. This algorithm takes n transactions and their C-SAGs G_l as the inputs. Besides, there are two global data structures: (1) a set M_l of access sequences as defined in Definition 4; and (2) a queue Q_{ready} of transactions that are ready for execution. At the beginning, all transactions that do not conflict with any other transaction proceeding themselves are inserted into the queue Q_{ready} (Line 1). For example, the transactions T_1 and T_2 in Fig. 4(a) are ready for execution initially, because all the state item values to be read by them can be retrieved from the latest snapshot S^{l-1}. Then, in the *execution phase*, transactions are popped from Q_{ready} (Line 4) and bound to a thread for parallel executions (Lines 6–17). With the executions of transactions, new writes are inserted into the access sequences of some state items (Line 16). When all writes that should be read by a transaction are completed, the validator pushes it into Q_{ready}, waiting for future execution.

Algorithm 1: Schedule generation for DMVCC

```
   Input: a block $B_l = \langle T_1, \dots, T_n \rangle$,
   a set of C-SAG $G_l = \{AG_1, \dots, AG_n\}$
   Data: a set $M_l$ of access sequences, a queue $Q_{ready}$ of ready
         transactions
   /* Grant state items locks to transactions only
      reading values from the snapshot                  */
1  $Q_{ready} \leftarrow Initialize(M_l)$
2  ▷ Execution Phase // Run in parallel
3  while executions for $T_1, \dots, T_n$ is not complete do
4      $T_i \leftarrow Q_{ready}.pop()$
5      if $T_i \neq null$ then
           // Run on a separate thread
6          $(code, pc) \leftarrow Get\_Code(T_i)$
7          $W \leftarrow \emptyset$ // map from state items to values
8          foreach $op \leftarrow GetOp(code, pc)$ do
9              if $op$ is SSTORE with key $I$ then
10                 $W[I] \leftarrow Val(I)$
11             else if $op$ is SLOAD with key $I$ then
12                 $Execute\_Read(M_l[I], W, T_i)$
13             else
14                 $pc \leftarrow ExecuteOp(op, M_l)$
               // Make existing writes visible to
                  later transactions (Algorithm 2)
15             if $AfterReleasePoint(AG_i, pc) = true$ then
16                 $W \leftarrow Early\_Write(W, G_l, pc)$
17             $pc \leftarrow NextPC(op, pc)$
18 ▷ Commit Phase // Run in parallel
19 if executions for $T_1, \dots, T_n$ complete then
20     Flush last write of every access sequence in $M_l$ to $StateDB$
         and make a new snapshot $S^{(l)}$.
```

Every transaction execution runs on a separate thread. First, each validator obtains the *code* and *pc* from the EVM opcode of the smart contract invoked by the transaction T_i (Line 6). Then a map W indexed by state items is initialized, to buffer temporary values for writes (Line 7). Next, the validator executes every operation $op \in code$ sequentially (Lines 8–17). In each iteration, if *op* is "SSTORE" (write operation), the validator writes $Val(I)$ to the buffer W. The values buffered in W will be made visible to other execution processes through the shared access sequences, when current process reaches a release point, i.e., can be visible early as explained later. If *op* is "SLOAD" (read operation), the validator reads the value of I from the buffer W, the latest snapshot S^{l-1}, or the access sequence $M_l[I]$ (Line 12). Here, T_i in $Execute\_Read()$ is used to determine which version should be read. Otherwise, the validator simply executes *op* according to its original semantics (Line 14). Finally, when execution reaches a release point, the write of a state item can be made visible earlier only if it will not be updated in the future. The function $Early\_Write(\cdot)$ is used for this purpose, which is detailed in Algorithm 2.

In the *commit phase*, when the executions of all transactions are finished, the validator flushes the results (i.e., the last write in every access sequence), to the *StateDB* and makes a new snapshot S^l (Line 20). Note that the two phases may run in parallel.

C. Early Write Visibility

Previous works [6, 10, 12, 22] synchronize writes at the transaction level: the writes of state items are made visible to other transactions only when they are committed or at least finished. With our more fine-grained statement-level analysis on state accesses, the written values can be made visible to the succeeding transactions much earlier, known as the *early write*

Algorithm 2: Early_Write

```
Input: write buffer $W$, C-SAG $AG_i$, program counter $pc$
Output: write buffer $W$
Data: the set $M_l$, the queue $Q_{ready}$
1  if remaining gas is sufficient then
2      foreach $I, Val(I)$ in $W$ do
3          if there is no write of $I$ in successor nodes then
4              $L_I \leftarrow M[I]$
               // Algorithm 3
5              $allowed, aborted \leftarrow Version\_Write(L_I, T_i, Val(I))$
6              $W.Delete(I)$
7              foreach $T_k$ in $allowed$ do
8                  $ready \leftarrow Grant\_Lock(T_k, I)$
                   // All locks of reads are collected, $T_k$ becomes ready
9                  if $ready = true$ then
10                     $Q_{ready}.Push(T_k)$
11             foreach $T_k$ in $aborted$ do
12                 $Abort(T_k, I, AG_i)$
13                 $ready \leftarrow Grant\_Lock(T_k, I)$
14                 if $ready = true$ then
15                     $Q_{ready}.Push(T_k)$
16 return $W$
```

visibility [23]. Yet, the downside of sharing premature writes is that it may incur more transactions aborted, for instance, when the writes are abandoned due to various exceptions. In such cases, a validator has to abort and re-execute the transactions that depend on the premature values. To reduce the cascading aborts, we make writes visible when the execution reaches a *release point* in the C-SAGs. A release point in a CFG is a node whose successors do not contain any abortable statement. A CFG may have multiple release points, and we only retain the earliest ones whose predecessors are not release points. The gas field of a release point gives an upper bound to the gas needed for the remaining instructions, which is used to judge whether the transaction can potentially be aborted due to the lack of gas. If there's enough gas and no abortable statement in remaining execution paths, we can conclude that the transaction will not be aborted and make its writes visible to other transactions safely. The gas estimation is done for C-SAGs since loops may not be unrolled for P-SAGs.

Algorithm 2 depicts the early write visibility procedure, called after a release point. First, the validator ensures that the remaining gas at the current node (labeled with pc) is greater than the estimated upper bound (Line 1). Then, for each $I \in W$, if there exists no write on I in the successors of the current node, the validator fills the value of I as an intermediate version in the access sequence L_I (Lines 3–5). When a transaction T_i acquires a lock on the state item, it means that the execution of it can read the correct version of the value from the access sequence. With the newly written value, the transactions depending on it (denoted by $allowed$), are enabled and gain the lock of I immediately. If a transaction in $allowed$ has yet collected all the locks required, it is then pushed into the queue Q_{ready}, waiting for being executed (Lines 7–10). If this write is not detected before, it will cause transactions that have read a stale version written by prior transaction to be aborted and re-executed (Lines 11–15). The details of the $Abort()$ function is elaborated in Section IV-E.

Algorithm 3: Version_Write

```
Input: Access sequence $L_I = \langle T_{p_1} : \alpha_{p_1}, \dots, T_{p_k} : \alpha_{p_k} \rangle$,
       transaction $T_i$, value of state item $Val(I)$
Output: $allowed$: a set of transactions to be granted lock, $aborted$:
        a set of transactions to be aborted
1  $allowed \leftarrow \emptyset$, $aborted \leftarrow \emptyset$
2  $t \leftarrow$ find out $t$ such that $p_t \le i < p_{t+1}$
3  for $j \leftarrow t+1$; $\alpha_{p_j} = \rho$ or $\alpha_{p_j} = \theta$; $j \leftarrow j+1$ do
4      if read operation $\alpha_{p_j}$ is completed then
5          $aborted \leftarrow aborted \cup \{T_{p_j}\}$
6      else
7          $allowed \leftarrow allowed \cup \{T_{p_j}\}$
8  if $t \notin \langle p_1, \dots, p_k \rangle$ then
9      $L_I \leftarrow \langle \dots, T_{p_{t-1}} : \alpha_{p_{t-1}}, T_i : \omega, T_{p_t} : \alpha_{p_t}, \dots \rangle$
10 else if $\alpha_{p_t} = \rho$ then
11     $L_I \leftarrow \langle \dots, T_{p_t} : \theta, \dots \rangle$
12 $Write\_Value(L_I, T_i, Val(I))$
13 return $allowed, aborted$
```

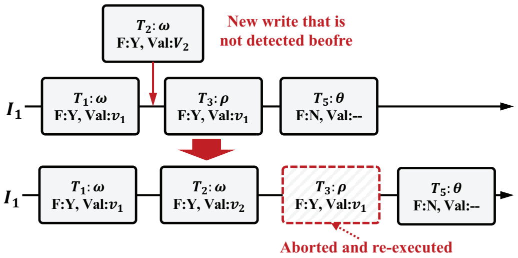


Fig. 5: Abort of transaction execution.

D. Write Versioning and Commutative Writes

Algorithm 3 presents how write versioning works on an access sequence. The procedure takes an access sequence L_I of the state item I, a transaction T_i, and the value $Val(I)$ as inputs. First, the validator finds out a proper location t in L_I, where the new write of $Val(I)$ should be placed (Line 2). With the arrival of a new write, the transactions after t, which have read a value of I, should be aborted, since they have read a stale value (Line 5). On the other hand, the transactions waiting for the new value are allowed to read (Line 7). Next, the access sequence is refined in case the analysis performed before is inaccurate, i.e., missing write operations. If there was no access to I in T_i, we insert a new write operation $T_i : \omega$ into L_I (Line 9). If I was only read by T_i, we change the operation to θ, i.e., both read and write (Line 11). Then, the validator writes value to L_I by setting the field "F" to "true" and copying $Val(I)$ to the field "Val" through the function $Write\_Value()$ (Line 12). Finally, the sets $allowed$ and $aborted$ are returned.

Commutative writes. Another strategy to further increase the parallelism is to bypass avoidable conflicts arising from *commutative writes*. Inspired by the previous works [10, 24], we notice that two transactions both incrementing the same state item without reading its original value are semantically commutative. Thus, they are not considered conflicting and both increments can be stored in any order. When reading the state item, the validator merges both increments to determine the final value. Fig. 6 illustrates the effects of early write visibility and commutative write, compared with the parallel execution in Fig. 4(b). Thanks to write versioning, transactions that are going to write the same state item do not conflict with each other and can be executed in parallel. As illustrated above, transactions T_2 and T_4 write the same state I_2 commutatively (denoted by $\overline{\omega}(I_2)$), thus the validator can execute them in parallel. Otherwise, T_2 and T_4 must be executed serially, resulting in Thread 3 staying idle. Also, because I_1 is made visible early by T_1, T_3 can be executed right after T_4 is finished.

E. Transaction abort

The abort of transactions happened in DMVCC can be classified into two categories: *deterministic abort* and *non-deterministic abort*. The deterministic abort is caused by some deterministic abort logic of smart contract statements. For example, when an *assert()* does not hold or the gas runs out, the execution is interrupted explicitly. These aborted transactions do not need to be re-executed because they follow the original semantics of the code. The non-deterministic abort occurs when there is some unpredictable situation that violates the deterministic serializability. This is inevitable since we cannot ensure the state access graph analysis is accurate completely. If the write on I does not appear in the access sequence (due to imprecision in program analysis), a new node will be inserted into the access sequence. For example, in Fig. 5, the write "$T_2 : \omega$" is not detected before and T_3 has read the version written by T_1. As "$T_2 : \omega$" is inserted into access sequence, the version v_1 read by T_3 becomes stale and T_3 should be re-executed.

Algorithm 4: Abort

```
   Input: Transaction $T_k$, state item $I$, complete access graph
          $AG_k$
   Data: the set $M_l$, the queue $Q_{ready}$
1  if $T_k$ does not gain the lock of $I$ then
2      return
3  else if $T_k$ in $Q_{ready}$ then
4      Remove $T_k$ from $Q_{ready}$
5  else if $T_k$ is running then
6      Stop execution for $T_k$
7  $Release\_Lock(T_k, I)$
   // Handle cascading aborts recursively
8  $\mathcal{W} \leftarrow AG_k.Get\_Visible\_Writes()$
9  foreach $I_i$ in $\mathcal{W}$ do
10     $L_{I_i} \leftarrow M_l[I_i]$
11     $allowed, aborted \leftarrow Version\_Write(L_{I_i}, T_k, null)$
12     foreach $T_j$ in $aborted$ do
13         $Abort(T_j, I_i, G_l)$
```

Algorithm 4 details the non-deterministic transaction abort. This algorithm takes a transaction T_k, a state item I that incurs the abort of T_k and the C-SAG set AG_k as inputs. The data are the same as the aforementioned algorithms. If the transaction T_k does not gain the lock of state I, the validator does nothing (Lines 1-2). If T_k is ready for execution, the validator removes it from the queue Q_{ready} (Lines 3-4). Otherwise, if T_k is running, the validator stops its execution. Then the validator releases the lock that has been granted to T_k (Line 7). Next, the validator tries to abort all transactions that have read the value written by itself. Specifically, the validator gets all writes $\mathcal{W}$ that already are visible to other transactions (Line 8). For each state I_i in $\mathcal{W}$, the validator resets the node of T_k in the access sequence L_{I_I} by inserting an empty write ($null$) and abort every transaction T_j that has read the version of I_i written by T_k recursively (Lines 9-13)).

Transaction aborts are caused by unexpected write operations. If the accuracy of the state access graph analysis is improved, the abort ratio can be reduced significantly. Yet, the presence of inaccurate analysis is inevitable since the logic of smart contract can be complicated. In such cases, the abort mechanism can still promise the deterministic serializability.

F. Correctness of DMVCC

To investigate the correctness of the DMVCC protocol, we show that it meets the deterministic serializability criteria, given in Theorem 1. Intuitively, our protocol ensures that if a stale value of a state item is read, the execution of that transaction will eventually be aborted and the incorrect results will be reverted. We first prove the following lemma.

Lemma 1. *For a block $B_l = \langle T_1, \ldots, T_m \rangle$, if an execution $\mathcal{E}_i$ of the transaction T_i reads a value of state item I that is stale or not finally committed, $\mathcal{E}_i$ will be aborted eventually.*

Proof. According to the assumption, let v_j written by T_j be the write (version) read by an execution $\mathcal{E}_i$. Then there are two cases: (1) v_j becomes stale later; (2) v_j becomes invalid, since T_j is aborted later. For the first case, a transaction T_k ($j < k < i$) should write a new version v_k into the access sequence L_I. If transaction T_k writes v_k before $\mathcal{E}_i$ happens, the transaction T_i will read the write v_k instead of v_j, since every transaction reads the latest write preceding it. If the transaction T_k writes v_k after $\mathcal{E}_i$ finishes, the transaction T_i should be aborted and re-executed according to the abort mechanism. For the second case, v_j becomes invalid when T_j is aborted, e.g., T_j makes v_j visible to T_i early, but is aborted as a result of running out of gas[3] or other unexpected reasons. At this moment, all writes performed by T_j will be rolled back and T_i should be aborted in cascade subsequently. Besides, commutative write does not affect the lemma. If transaction T_i is going to read a state item I, the validator merges all increments to a complete state. Therefore, we can conclude that T_i reads the latest normal write preceding it.

In summary, the lemma holds for the execution of every transaction. □

Theorem 1. *Given block $B_l = \langle T_1, \ldots, T_m \rangle$, the schedule generated by DMVCC produces the same results as the serial execution.*

[3] Albeit the efforts in making accurate gas estimations, running out of gas may still happen in some extreme situations.

Proof. We prove the correctness of this theorem by induction. Let execution $\mathcal{E}_i^*$ be the last execution for T_i. Obviously, the executions $\mathcal{E}_1^*, \ldots, \mathcal{E}_i^*$ do not read any write generated by $\mathcal{E}_{i+1}, \ldots, \mathcal{E}_n$. Therefore, $\mathcal{E}_{i+1}, \ldots, \mathcal{E}_n$ do not affect the deterministic serializability of $\mathcal{E}_1^*, \ldots, \mathcal{E}_i^*$. We assume the parallel executions $\mathcal{E}_1^*, \ldots, \mathcal{E}_i^*$ meets deterministic serializability. Then the execution $\mathcal{E}_{i+1}^*$ for T_{i+1} will be committed if and only if $\mathcal{E}_{i+1}^*$ does not read any stale or invalid value of every state item according to Lemma 1. It means $\mathcal{E}_{i+1}^*$ reads the latest writes for all state items generated by $\mathcal{E}_1^*, \ldots, \mathcal{E}_i^*$. Therefore, the parallel executions $\mathcal{E}_1^*, \ldots, \mathcal{E}_{i+1}^*$ have the same effect as the serial executions for $\langle T_1, \ldots, T_{i+1} \rangle$. In summary, the theorem holds for $\mathcal{E}_1^*, \ldots, \mathcal{E}_n^*$. □

V. Implementation and Evaluation

In this section, we provide the implementation details of DMVCC and discuss its experimental evaluation.

A. Implementation

We integrated DMVCC into the Go Ethereum (Geth) platform [25] and the contract-related analyses were implemented based on the Slither tool [13, 26]. If the smart contract source code is unavailable, we could still construct CFGs from bytecode, so that the SAG can be built from the CFG of every contract. The DMVCC protocol was implemented in Go with about 4,000 lines of code. Since the EVM implementation in Geth does not support multi-threading, we modified the validator code to create multiple EVM instances in advance. When a transaction becomes ready, the customized validator binds an EVM instance to a CPU core to execute the transaction accordingly.

In practice, the persistent memory storage of a smart contract is represented as an array of 2^{256} slots, as opposed to the simplified state view used in Section III. During compilation, each Solidity variable is mapped to one or more slot(s), according to a predefined addressing rule [27]. The values of the slots are persisted to a special form of Merkle tree, called the Merkle Patricia Tree (MPT). The MPT generates a snapshot of the current state automatically, at the end of execution, for each block. Sequentially, DMVCC treats each slot as an independent state item. The state access graphs were built based on the CFGs generated using Slither.

The access sequences are implemented as buffers between the EVMs and the underlying MPT to save all writes generated by executions for transactions in the current block. The executions for transactions in B_l fill writes into the access sequences of various state items through write versioning and read proper versions from them. When all executions finish, we flush the write operations cached in the access sequences into MPT by inserting the last write of each access sequence into the MPT.

B. Experiment Setup

Comparisons. We compared DMVCC with the default serial execution implemented by the original EVM and two other existing parallel execution solutions: (1) a DAG-based approach [6]; and (2) an OCC-based approach [10, 14]. DAG-based approach uses a *Directed Acyclic Graph* (DAG) to depict the conflicts between transactions and then allows non-conflicting transactions to be executed in parallel. This approach is similar to ours, but it does not tolerate incorrect analysis of the read/write set and includes the write-write conflicts that are eliminated by DMVCC. The OCC-based approach is introduced in Section II-B. For a fair comparison, we implemented both approaches on the Geth platform although their implementations are on different platforms.

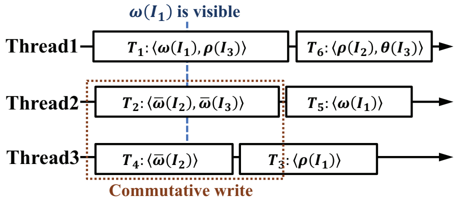

$\omega(I_1)$ is visible
Thread1
$T_1: \langle \omega(I_1), \rho(I_3) \rangle$
$T_6: \langle \rho(I_2), \theta(I_3) \rangle$
Thread2
$T_2: \langle \bar{\omega}(I_2), \bar{\omega}(I_3) \rangle$
$T_5: \langle \omega(I_1) \rangle$
Thread3
$T_4: \langle \bar{\omega}(I_2) \rangle$
$T_3: \langle \rho(I_1) \rangle$
Commutative write

Fig. 6: Scheduling with early write visibility and commutative write for transactions in Fig. 4.

Datasets. To test the performance of different algorithms on real-world workloads, we synchronized the transactions from the mainnet of Ethereum as our test data. Our experiment focuses on the period between Jan 1, 2022 and April 30, 2022 (769,020 blocks in total). There are 122 million transactions in these blocks, and about 84 million of them (69%) made contract calls, to about 61,392 different smart contracts and the rest are non-contract transactions. ERC20 tokens (e.g., token distributions, airdrops) accounted for 60% of the traffic, Decentralized Finance (DeFi) applications made up 29%, while games and collectibles (NFTs) triggered another 10% of the transactions. For each non-contract transaction, it is trivial to infer its read/write sets from its inputs and there is no need to construct state access graph for it. We simply add the non-contract transactions as constraints when calculating the schedule, without any change to the schedule generation algorithm.

Testbed. Our experiments were conducted on a Ubuntu 18.04.3 LTS desktop equipped with an Intel Core i7 16-core and 32GB memory. We simulated scheduling the transactions on a set of threads (up to 32). Every evaluation result we report is the average of four independent runs.

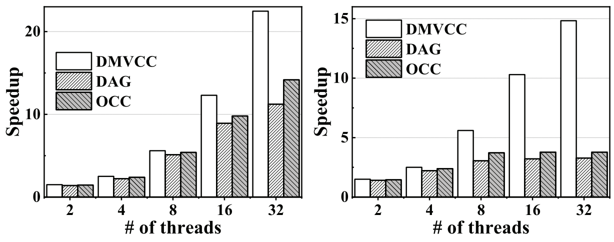
C. Experiment Results

To explore the capability of our proposed approach, we evaluated DMVCC to answer the following research questions.

- **RQ1**: How well do the parallel execution results of DMVCC meet the deterministic serializability criteria?
- **RQ2**: How much speedup can DMVCC achieve over the serial execution and how does it compare with other existing approaches?
- **RQ3**: How efficient is DMVCC in a real-world blockchain environment?

We now discuss the experiment findings in details.

Results for RQ1. Theorem 1 states the correctness of DMVCC theoretically. To validate the correctness of our implementation,



(a) Low contention

(b) High contention

Fig. 7: Speedup of all parallel execution approaches. The x-axis shows the number of threads, and y-axis shows the speedup achieved.

we compared the state snapshots produced by DMVCC with that of the serial execution. Since Ethereum maintains its state snapshot using Merkle trees [2], we could easily test if two state snapshots are identical by comparing the values of their Merkle roots. The Merkle root value of a block can be obtained after all the transactions of the block are executed. We tested a total number of 121,210 blocks, carrying 22,557,724 transactions, since the time needed to execute all blcoks serially is prohibitive. The comparison always results in a matching value of the Merkle roots for every block.

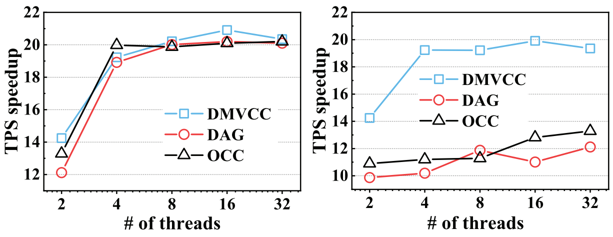
Results for RQ2. To measure the effectiveness of DMVCC, we first report its effective speedup against the serial execution of the original EVM implementation. In this experiment, we evaluated only the performance of pure EVM execution without taking the impact of consensus into account. For every non-contract transaction, we directly transferred Ethers without a need to start an EVM instance. We repacked transactions into blocks randomly, each of which contains 1,000 transactions, because each original block contains fever transactions. For each block, transactions were executed with up to 32 threads in parallel. Comparing with the serial execution baseline over all blocks, the average speedup of DMVCC with 32 threads is 21.35× as shown in Fig. 7(a), while the average speedups of DAG and OCC are 11.04× and 13.86×, respectively. Specifically, DMVCC is able to save 30 to 40s execution time per block cycle. These results show that DMVCC is able to exploit more parallelism among the transactions. This is because two transactions with write-write conflict, which is eliminated in DMVCC, cannot be executed in parallel in the DAG-based approach. Compared with the OCC-based approach, the abort rate of DMVCC is less than 2% and DMVCC reduces 63% unnecessary transaction aborts by detecting potential conflicts between transactions in advance. Besides, when the number of threads is small, the performance difference between the three approaches is not significant, when they all exploit the capacity of threads. As the number of threads increases, the speedup for DAG and OCC grows at a lower rate, because some threads may stay idle during execution for the DAG-based and OCC-based approaches.

To verify the effectiveness of the early write visibility and commutative write features, we simulate some high-contention blocks, in which there are more conflicts between transactions. We selected 1% of the smart contracts as the hot contracts and each transaction has a 50% probability to access the hot accounts, which is used by many previous works [28] in evaluating performance of transaction executions. Fig. 7(b) shows the speedup of all approaches under a skewed workload with high contention. The average speedup of DMVCC on all blocks is 13.73× with 32 threads, while the average speedup of DAG-based and OCC-based approaches are 3.05x and 3.48×, respectively. With high contention, the speedup of DAG and OCC declines significantly because the inherent parallelism between transaction is low. DMVCC, instead, reduces the conflicts through commutative write and allows transactions to be executed earlier with early write visibility. As a result, DMVCC achieves much better scalability when the chance of conflicts is high.

Results for RQ3. To evaluate the efficiency of all approaches in a real-world blockchain environment, we built a micro Ethereum testnet with 20 validators (miners). To simulate the traffic of the Ethereum mainnet, we adjusted the mining difficulty to ensure that a new block is mined in approximately every 12 seconds, which is on a par with the Ethereum mainnet. Since the major bottleneck of the current Ethereum blockchain still lies in mining, a block can only be packed with about 180 transactions. The advantage of parallel execution in such a setting is not obvious. To simulate a blockchain environment with optimized mining protocols and increased block sizes, e.g., Ethereum 2.0 [29] and EOS [30], we adjusted the gas limit to allow each validator to pack up to 10,000 transactions per block. Besides, the setting of high-contention is the same as that in **RQ2** except for the block size. Under such a setting, the overhead shifts to transaction executions, and parallel executions start to show significant improvements. The execution time of the tested transactions varies substantially, ranging from sub-milliseconds to tens of milliseconds.

As is shown in Fig. 8, the throughput speedup increases as the number of threads grows. In a low-contention setting, the average throughput speedup of DMVCC over all blocks is 19.79× with 32 threads, and the speedup of DAG and OCC is similar. Transaction execution in such a setting is not a bottleneck and the speedup is linear to the number of transactions per block. Besides, the overhead for P-SAG and C-SAG construction does not affect the execution performance of DMVCC, because these processes are performed offline, before the execution.

However, when the contention becomes high, as depicted in Fig. 8(b), the transaction execution becomes the bottleneck. With more threads, the increase in terms of throughput speedups achieved by DAG and OCC are small. This is because DAG and OCC could only finish executing 60% of the transactions that DMVCC could finish within the same time period. DMVCC, instead, can finish executing 10,000 transactions with 8 threads within 12 seconds (a mining cycle). Such high-contention cases are common when an Initial Coin Offering (ICO) is launched, where almost all transactions in the recent blocks access the same ICO contract. Moreover, we also adjusted the mining difficulty, allowing validators to generate a block in every second. Then, the transaction execution becomes the main



(a) Low contention (b) High contention

Fig. 8: Throughput speedup for blockchain of all parallel execution approaches.

bottleneck, and the speedup achieved in throughput is closely related to the execution.

D. Threats to Validity

Our selected transactions may not be representative of all traffic patterns on Ethereum, or other blockchain platforms. We mitigated this by selecting a large number of transactions covering a long period of time. Our implementation of DMVCC may contain bugs. We rigorously tested our tool for a prolonged period of time and fixed many bugs along the way. Our implementation of the DAG and OCC approaches may not accurately replicate their original proposals. We deployed some smart contracts with the same logic on both our implementation and the original platforms, and then compared the obtained results carefully to ensure the correctness.

VI. Related Work

Various types of techniques have been proposed to improve the performance of blockchains, including parallel transaction execution and sharding that are closely related to our work. Our work is also related to program analysis techniques applied to smart contracts.

Parallel transaction execution. The parallel execution of blockchain transactions has been extensively studied in recent years [7, 20]. Zhang & Zhang [7], instead of using a dependency graph, includes each transaction's write set in the block and allows validators to use them in detecting conflicts. Hyperledger Fabric [21] and its variants [15, 28] employ an *Optimistic Concurrency Control* (OCC) strategy to support parallel execution under a novel *execute-order-validate-commit* (EOVC) paradigm. However, the validation for transactions is performed sequentially, which may become the bottleneck of the system. Most of the proposed techniques are protocol-breaking, in the sense that they modify the block structure and the execution semantics. In contrast, our approach remains compatible with the existing implementation of Ethereum. The deterministic concurrency is also well studied in databases [31, 32]. The approach in [31] still needs an accurate analysis on read/write sets of transactions and may result in high abort rate when the contention between transactions is high. Sparkle [32] also employs an OCC-based currency protocol and applies the abort mechanism to ensure determinisitc serializability.

Sharding. Another way to improve blockchain performance is through sharding, which splits the set of nodes into a number of smaller committees. Incoming transactions can then be processed by different committees in parallel. This has been an popular research topic recently, in both industry [33, 34] and academia [35–38]. Many of these works focus exclusively on applying sharding on the simplest kind of transactions, i.e., *user-to-user transfers of digital funds*, while ignoring more complex smart contract transactions [36–38]. Existing proposals targeting smart contract transactions impose heavy restrictions on the contract-manipulating transactions, e.g., processing all such transactions in a specialized shard [35, 39]. Other solutions assume that a complex cross-shard communication protocol reconciles possible conflicts [34, 40], or adopt a contract design very different from that of Ethereum [2].

Smart contract program analysis. There have been a large number of static [41–43] and dynamic [44–46] analysis techniques developed for smart contracts. Most of these techniques focus on security issues. For example, Oyente [47] is one of the earliest static analysis tools to detect security vulnerabilities such as reentrancy, and Slither [13] is used to perform taint analysis to find information flow related vulnerabilities. Similarly, fuzzing [44–46] and model-based testing [48, 49] have been explored to discover common security issues. Moreover, Pîrlea et al. [24] proposed analyses on data ownership and commutativity of operations, which are used to accelerate executions for cross-shard smart contract transactions. In this paper, we leverage precise static and dynamic program analysis to enable more fine-grained state accesses and improve the performance of parallel execution.

VII. Conclusion

With the evolution of consensus protocols for public blockchains, the execution efficiency is becoming the new bottleneck of the entire system, driving the need for better transaction parallelization. This paper introduces a novel scheduling framework, DMVCC, which improves parallelism for high-contention transactions with more fine-grained state accesses. DMVCC supports write versioning, which helps avoid write-write conflicts, and early write visibility, which makes writes visible to other transactions, as soon as there is no risk of abort. The evaluation results demonstrate that DMVCC maintains deterministic serializability, and significantly outperforms other modern parallel execution techniques.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme, National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[3] "Blockchain Technology Use Cases in Financial Services," http://blog.deloitte.com.ng/5-blockchain-use-cases-in-financial-services/, 2017.

[4] "IBM Blockchain for Supply Chain," https://www.ibm.com/blockchain/supply-chain/, 2020.

[5] "Blockchain: Opportunities for Health Care," https://www2.deloitte.com/us/en/pages/public-sector/articles/blockchain-opportunities-for-health-care.html, 2018.

[6] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1337–1347.

[7] A. Zhang and K. Zhang, "Enabling concurrency on smart contracts using multiversion ordering," in *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 2018, pp. 425–439.

[8] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, and A. C.-C. Yao, "A decentralized blockchain with high throughput and fast confirmation," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 515–528.

[9] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "Ohie: Blockchain scaling made simple," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 90–105.

[10] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, "Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts," *arXiv preprint arXiv:2201.03749*, 2022.

[11] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," *arXiv preprint arXiv:1901.01376*, 2019.

[12] "FISCO-BCOS," http://fisco-bcos.org/, 2020.

[13] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[14] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain meets database: Design and implementation of a blockchain relational database," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1539–1552, 2019.

[15] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 105–122.

[16] D. Reijsbergen and T. T. A. Dinh, "On exploiting transaction concurrency to speed up blockchains," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 1044–1054.

[17] "Solidity," https://docs.soliditylang.org/, 2022.

[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[19] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[20] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 83–92.

[21] E. Androulaki *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[22] M. Fang, Z. Zhang, C. Jin, and A. Zhou, "High-performance smart contracts concurrent execution for permissioned blockchain using sgx," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1907–1912.

[23] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, 2017.

[24] G. Pîrlea, A. Kumar, and I. Sergey, "Practical smart contract sharding with ownership and commutativity analysis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1327–1341.

[25] "Go Ethereum," https://github.com/ethereum/go-ethereum, 2022.

[26] "Slither tool," https://github.com/crytic/slither, 2022.

[27] "Solidity storage layout," https://docs.soliditylang.org/en/v0.6.8/internals/layout_in_storage.html, 2022.

[28] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 543–557.

[29] "Ethereum 2.0," https://ethereum.org/en/upgrades/beacon-chain/, 2022.

[30] B. Xu, D. Luthra *et al.*, "Eos: An architectural, performance, and economic analysis," *Retrieved June*, vol. 11, p. 2019, 2018.

[31] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 70–80, 2010.

[32] Z. Li, P. Van Roy, and P. Romano, "Sparkle: Speculative deterministic concurrency control for partially replicated transactional data stores," 2018.

[33] E. A. Stoica and D. M. Sitea, "Blockchain disrupting fintech and the banking system," in *Multidisciplinary Digital Publishing Institute Proceedings*, vol. 74, no. 1, 2021, p. 24.

[34] A. Skidanov and I. Polosukhin, "Nightshade: Near protocol sharding design," *URL: https://nearprotocol. com/downloads/Nightshade. pdf*, p. 39, 2019.

[35] H. Dang, T. T. A. Dinh, D. Loghin *et al.*, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 123–140.

[36] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[37] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 17–30.

[38] M. Zamani and M. Movahedi, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 931–948.

[39] K. Amrit, "Provisioning sharding for smart contracts: A design for zilliqa," 2018.

[40] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," *ACM SIGPLAN Notices*, vol. 44, no. 1, pp. 2–15, 2009.

[41] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[42] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy *et al.*, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[43] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[44] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[45] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1795–1809, May 2022.

[46] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," 2020, pp. 1398–1409.

[47] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[48] Y. Liu, Y. Li, S.-W. Lin, and Q. Yan, "ModCon: A model-based testing platform for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1601–1605.

[49] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. New York, NY, USA: ACM, Jul. 2022, pp. 716–727.