Enabling Concurrency on Smart Contracts Using Multiversion Ordering

An Zhang(✉) and Kunlong Zhang

School of Computer Science and Technology, Tianjin University, Tianjin, China
{zhangan,zhangkl}@tju.edu.cn

Abstract. Blockchain-based platforms, such as Ethereum, allow transactions in blocks to call user-defined scripts named *smart contracts*. In the blockchain network, after being generated by a miner, a block will be validated many times by the peers who accept it. Hence by enabling concurrency on smart contracts, especially validation, we can improve the efficiency and the throughput of those platforms.

By introducing multiversion transaction ordering, this paper presents a concurrent scheme called *MVTO* to run smart contracts concurrently. First, the miners are able to use any concurrency control technique to discover a conflict-serializable schedule. Then, validators use MVTO to verify the block by replaying this schedule concurrently and deterministically. The evaluation shows that this mechanism achieves approximately 2.5x speedup in the block validation using a thread pool with 3 threads.

Keywords: Blockchain · Smart contract · Concurrency
Multiversion transaction ordering

1 Introduction

Platforms like Ethereum are essentially instances of distributed Byzantine-fault-tolerant database. They are built on a decentralized, peer-to-peer network where peers do not fully trust each other. Generally, there are two kinds of peer nodes in the network: miner and validator. We briefly describe their work as follows: Miners repeatedly collect transactions in the network and package them into new *blocks*. When creating a new block, the miner incorporates the cryptographic hash of the preceding block of the new block, *i.e.*, the most recent block in its local storage, into the header of the newly generated block. The cryptographic hash in each block's header acts as the pointer to its preceding block and thus form a chain of blocks, called *blockchain*. The newly generated block is then published to other validators. Validators follow a consensus protocol to decide whether to accept a newly received block or not and how to synchronize with other peers to reach consensus of blockchain states across the network. In a word, the blockchain is a shared immutable database for recording the history of transactions.

Y. Cai et al. (Eds.): APWeb-WAIM 2018, LNCS 10988, pp. 425–439, 2018.
https://doi.org/10.1007/978-3-319-96893-3_32

Ethereum introduces *smart contract* into blockchain. A smart contract is a collection of code (its functions) and data (its states) which are stored on the blockchain with a unique account and address [1]. A contract will execute when it is triggered by a transaction sent to its address. The functions called can be written in several Turing-complete languages such as Solidity [2]. Thus miner will charge fees from target contract's account for every computational step to ensure that the execution will finish. This fee refers to the *gas* in Ethereum.

Smart Contracts execute on blockchain in two scenarios.

1. Mining: When a miner proposes a new block, it starts to execute contracts according to the order of the transactions in the block. The merkle root of final states is then stored in the block.
2. Validating: When a validator receives a new block, it re-executes smart contract with the exact same order adopted by the miner when this block was generated. Then the validator checks the consistency of the resulting states by using merkling techniques.

Despite having the advantages such as tamper-proof and Byzantine-fault-tolerant provided by blockchain, smart contract platforms suffer from the limitation of throughput. This limitation is partly prompted by the lack of concurrent mechanisms in existing smart contracts designs. When miners and validators deal with a block, they execute smart contracts serially to produce a deterministic result. Furthermore, Ethereum is planning to change its consensus protocol from *proof of work* (POW) to an energy saving protocol called *proof of stake* (POS). After switching to POS, Ethereum can significantly save time originally needed in the POW phase. In this trend, Ethereum can execute smart contracts that are more complicated and time-consuming. However, one needs to apply concurrency to fully exploit those saved computational powers.

There are three reasons why smart contracts can not employ naive concurrent solution. First, a smart contract can be called several times by different transactions during a block's execution and therefore race conditions may occur when those calls of the same contract execute in parallel. Second, smart contracts need to execute transactionally. In other words, if Ethereum executes multiple transactions concurrently, it must produce a *conflict-serializable* schedule where the final states can be produced by a serial schedule[1]. Third, validating a block requires deterministic execution which can not be provided by naive concurrency approach.

In the mentioned two scenarios where smart contracts are executed, the computational power spent on these scenarios is unbalanced. A block only gets executed one time when created by miners. If this block is accepted, every node in the network will validate this block. Thus, improving the efficiency of block validation is more important than block generation in mining phase.

We propose a concurrent scheme for smart contracts. In this scheme, when a miner proposes a new block, it can employ any concurrent control technique, such

[1] A serial schedule is a schedule where transaction are executed serially and do not interleave each other.

as 2PL or timestamp, as long as it can produce a *conflict-serializable* schedule. During the execution, the miner needs to record the *write set* of every transaction in the block, *i.e.*, the set which contains the data items that the transaction tries to write. After the miner finishes the block's execution, it stores the write sets into the block. Then, the miner adjusts the transaction order of the block to match the serial order of the resulting schedule and publish the new block. Before the validator executes the newly received block, it constructs a "write chain" on the conflicting data items using the write sets and the transaction order in the block. The "write chain" pre-determines the contention relationships among block's transactions and the priority of these transactions. The proposed mechanism, called *multiversion transaction ordering* (MVTO), uses "write chain" to resolve conflicts at runtime and then produces deterministic results. Meanwhile, by using the multiversion technique in the "write chain", MVTO further reduces the conflict at runtime.

This paper makes following contributions:

1. a scheme to run smart contracts called by transactions in a block concurrently where miner can employ any concurrent control technique as long as it produces conflict-serializable schedule.
2. a multiversion concurrency control mechanism to validate a block concurrently and deterministically. The evaluation shows that this mechanism achieves approximately 2.5x speedup in the block validation using a thread pool with 3 threads.

The rest of the paper is organized as follows. Section 2 introduces a simplified smart contract model and the notions used in this paper. Section 3 presents the details of the proposed mechanism called MVTO. Section 4 proves the correctness of MVTO. Section 5 illustrates the experiments and summarizes the results. Section 6 reviews the related work. Section 7 concludes the paper.

2 Background

This paper uses a simplified smart contract model to illustrate the proposed mechanism. This section will introduce the notions that related to this paper.

2.1 Smart Contract

Smart contracts can be seen as a collection of self-defined states maintained in blockchain and functions that manipulate these states.

A simplified proxy ballot contract is shown in Fig. 1. The contract defines two persistent states: voters (line 6) and proposals (line 7). "Proposals" is an array of "Proposal". Each "proposal" contains the number of votes it owns (line 3). The "voters" maps a unique memberID to the data structure "Voter" (line 2). Voters can vote to a specific proposal and they can only vote once.

Client firstly collects votes to the same proposal. Then it calls the function "proxyVote" (line 10–16) to cast these votes by sending a transaction to this

```
1  contract ProxyBallot {
2      //define Voter<int voteTo, int weight, int voted>
3      //define Proposal<int voteCount>
4
5      //self-define state
6      mapping(memberID -> Voter) voters;
7      Proposal[] proposals;
8
9      /// Cast votes to proposal $(toProposal) by Proxy.
10     function proxyVote(uint[] voters, uint8 toProposal) {
11         for(int senderID in voters){
12             Voter sender = voters[msg.sender];
13             if (sender.voted || toProposal >= proposals.length) throw;
14             sender.voted = true;
15             sender.voteTo = toProposal;
16             proposals[toProposal].voteCount += sender.weight;
17         }
18     }
19     //Other functions ...
20 }
```

Fig. 1. Simplified ballot smart contract

ballot contract's address. Voters cast their votes by adding their weight to target candidate's "voteCount". Most of those state changes are vulnerable to race conditions and may result in inconsistent states. Furthermore, functions can use **throw** statement to handle exceptions such as double voting (line 13). The **throw** statement can abort the contract, discarding the transient variables and undoing any state changes.

2.2 Data Action

A data action is a primitive operation (read or write) on a state's data item [3]. For example, in Fig. 1, voters cast their votes by performing an update action[2] on the target proposal in the state variable "proposals" (line 15). We refer to t_i as the i-th transaction in block. Notation $r_i(x)$ and $w_i(x)$ are the read and write action on data item x executed by t_i respectively.

2.3 Conflict

Two data actions *conflict* with each other if executing them in either order yields different results. For example, there are two situations where two actions conflict in the single-version concurrency control scenario:

1. Two actions of the same transaction, *e.g.*, $r_i(x)$ conflict with $r_i(y)$.
2. Two actions of different transactions and one of them is a write action, *e.g.*, $w_i(x)$ conflict with $r_j(x)(i \neq j)$ and $w_i(x)$ conflict with $w_j(x)(i \neq j)$.

2.4 Schedule

A *schedule* is the sequence of data actions that transactions actually performed. A *serial schedule* is a schedule that meets the following conditions: (i) the actions that belong to the same transaction preserve the order in transaction; (ii) the

[2] The update action can be divided into a read and a write action.

actions that belong to different transactions don't interleave with each other. For example, a serial schedule is shown in Example 1.

Example 1 (serial schedule).

$$r_1(x); w_1(x); r_1(y); w_2(x); w_2(y); r_3(x).$$

Conflict-Serializable Schedule. Two schedules are *conflict-equivalent* if they can turn into each other by swapping adjacent non-conflicting data actions. A schedule is *conflict-serializable* if it is conflict-equivalent to a serial schedule. The *serial order* of a conflict-serializable schedule is the transaction order of the serial schedule which it conflict-equivalent with.

Non-conflicting actions can be fully parallelized because the results won't change. However, swapping the execution order of conflict actions might change results and violate the correctness of transactional execution. In Example 2, we can observe that conflict-serializable schedules with the same serial order will produce the same results for each action that belongs to them.

Example 2 (conflict-equivalent). Following schedules are conflict-equivalent with the serial schedule in Example 1 and therefore have the same serial order $t_1 < t_2 < t_3$.

- $r_1(x); w_1(x); w_2(x); r_1(y); w_2(y); r_3(x);$
- $r_1(x); w_1(x); w_2(x); r_1(y); r_3(x); w_2(y);$

Hence, in order to ensure the correctness, smart contracts need concurrency control mechanism to produce conflict-serializable schedule.

Recoverable Schedule. When a transaction writes a data item, the write action is tentative and may be reverted if the transaction aborts. For the schedule "$w_2(x); r_3(x)$", $r_3(x)$ reads the value of x which is previously written by $w_2(x)$. The effect of $r_3(x)$ will not be recoverable and become a dirty data if t_2 aborts after t_3 commits. We refer to t_2 as a *read-from transaction* of t_3.

A schedule needs to be recoverable to prevent this dirty data issue. A *recoverable schedule* is a schedule where each transaction commits only after all its read-from transactions have committed. In this way, the changes made by the transaction that read dirty data are still revertable because the transaction must wait for its read-from transactions to commit.

2.5 Multiversion Concurrency Control

When using single-version concurrency control methods, transactions may have to abort because of non-serializable action. This paper employs multiversion concurrency control (MVCC) to reduce conflict. Instead of overwriting the data item when performing write action, MVCC preserves each version of write actions to avoid aborting transactions due to reasons like *read too late*. Since result of each

write action is recorded along with its timestamp, a read action can know which version it should read. We refer to the version of the data item x written by transaction t_i as notation x_i.

In MVTO, multiversion reduces the conflict among the data actions. Write actions on the same data item don't conflict because they write to different versions. The conflict happens between the read action and write action on the same version of the data item where the read action must wait for the write action finishes.

3 Proposed Mechanism

3.1 Basic Idea

Mining Phase. In mining phase, since correctness of smart contracts' execution requires conflict-serializability, any concurrency control mechanism used in DBMS can be applied to produce a conflict-serializable schedule. The block generating process can be divided into two steps: (i) contracts' execution and (ii) consensus protocol. After block's execution (step 1), the miner can acquire the serial order of transactions according to the produced schedule. Then, the miner adjusts the transaction order of the block to match the serial order. At this point, the miner can compute the hash of the reorganized block and step into consensus protocol (step 2). Figure 2 illustrates this block's reorganization process. In this example, the original order, which is decided by the miner, is $t_1 < t_2 < t_3$ (as shown in the left-hand figure). After the execution, the resulting schedule shows that the serial order is $t_1 < t_3 < t_2$ and therefore the miner reorganizes the block according to the serial order (as shown in the right-hand figure).

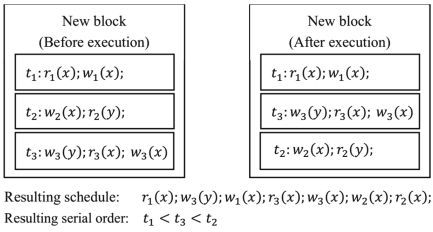


Fig. 2. When mining: reorganize transactions according to schedule

Validation Phase. In the concurrent scenario, verifying the transactions that call smart contracts need three conditions to ensure the correctness:

1. Conflict-serializability: the resulting schedule of validation must be *conflict-serializable*.
2. Schedule replayability: the resulting schedule must be *conflict-equivalent* with the schedule produced by the miner of this block.

3. Execution recoverability: the resulting schedule must be *recoverable* because the transactions can abort at any time.

We refer to the proposed mechanism as *multiversion transaction ordering* (MVTO). MVTO is similar to timestamp concurrency control because they both decide the presumptive transactions' serial order. Timestamp concurrency control presume the serial order using timestamp and abort those transactions which violate the presumptive order. Timestamps are assigned at runtime, which means the presumptive serial order is determined at runtime and the final order may change due to the abortion of transactions. On the contrary, MVTO presumes the serial order to be the transaction order[3] of the block before the execution so that it can validate deterministically. Therefore the validator can know the priority of transactions before the smart contracts' execution. Hence, MVTO can make sure that the data actions never violate the presumptive order.

In order to achieve such functionalities, MVTO needs transactions to provide their *write set* to the validation. Transaction's write set is the set which contains the data items that this transaction will write. Since smart contracts are written in Turing-compete languages, it is impossible in general to statically determine which data actions a smart contract will perform. However, every smart contract called in the block has already been executed when the block is created by the miner. Since mining phase and validation phase share the same initial states from the common blockchain history, these two phases also share the same write set of each transaction as long as their final serial order are identical. Therefore, mining phase can provide the write set of transactions to validation phase by recording all the write actions it has performed. By using these write sets, the validator can pre-determine all the versions that a data item will have and resolve the conflicts at runtime by letting the transaction with smaller index in the presumptive serial order execute first. Hence, the conflict-serializability and replayability of schedule can hold.

When the transaction t_i reads the data item's value which is previously written by another transaction t_j in this block, t_i will record the handler of t_j as its *read-from transaction*. In order to produce recoverable schedules, t_i cannot commit until all its *read-from transactions* has committed.

3.2 Data Structure

Transaction. The transaction in the block has four possible status: "Init", "Active", "Aborted" and "Committed". "Init" is the initial status of transaction. "Active" means there is a thread executing the transaction. A transaction are "Committed" when it completes all its tasks. A transaction will be "Aborted" when it uses up the *gas* of the contract or **throw** other exceptions. We refer to $status(t_i)$ as the status of t_i.

[3] This transaction order is also the serial order in the mining phase.

Read Set and Write Set. Transactions use *read set*, *i.e.*, the set of their read-from transactions, to ensure their recoverability. Read set is constructed at runtime and will be initialized into an empty set if the transaction restarts. After read action $r_i(x)$ reads the value written by another transaction t_j in the block, a tuple like $\langle t_j, x_j^i \rangle$ is inserted into read set of t_i. In this tuple, t_j is the read-from transaction of t_i and x_j^i is the value which t_i read from t_j. Note that we use x_j^i to indicate x_j^i may be inconsistent with x_j at the time t_i tries to commit, this will be discussed later. We refer to $rs(t_i)$ as the read set of t_i.

Every transaction has a *write set* that contains all the write actions it will perform. The write set of a transaction is generated during the first execution when the block is created by the miner. We refer to $ws(t_i)$ as the write set of transaction t_i. The elements in write set are data locators for the data items. With the write set, Validator can determine the conflict relation among transactions on the specific data items and construct the "write chain" for these data items accordingly.

Write Chain on Data Item. By using the write set, validator can construct *write chain* on every data item that will be accessed by the transactions in the block.

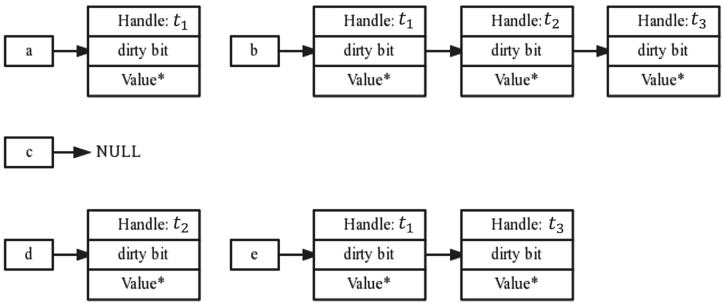
Figure 3 shows this write chain under an example. There are 5 data items (a, b, c, d, e). A validator receives a new block containing 4 transactions (Fig. 3a). According to the data actions of these 4 transactions, their write sets is shown in Fig. 3b.

$t_1 : r_1(a); w_1(a); r_1(b); w_1(b); r_1(e); w_1(e);$
$t_2 : r_2(b); w_2(b); r_2(d); w_2(d); r_2(d);$
$t_3 : r_3(e); w_3(e); r_3(b); w_3(b);$
$t_4 : r_4(a); r_4(b); r_4(c); r_4(d); r_4(e);$

(a) Transactions

$ws(t_1) : \{a, b, e\}$
$ws(t_2) : \{b, d\}$
$ws(t_3) : \{b, e\}$

(b) Write set



(c) Write chain

Fig. 3. Extended data structure on data item

Before the validator replays those transactions, it uses the write sets to construct the write chain. As shown in Fig. 3c, the write chain contains all the versions the data item will have during the block's execution. Elements in the write chain of a data item are 3-tuples: $\langle handle, value, write\ bit \rangle$, each of them represents a version of this data item. These elements are ordered by the index of their writer transaction, *i.e.*, the presumptive serial order. The "handle" is the reference to the transaction which is going to write the value into this element. Read actions will use this handle to access the index and the status of the writer transaction. When the transaction writes its version, it writes into the "value" of the element and set the "write bit" to **true**. For the read actions that want to read this version, the "value" is readable only when "write bit" is **true**. Otherwise, they have to wait for the writer transaction to write this version. We refer to $\langle t_i, x_i, wb(x_i) \rangle$ as the *target element* of action $w_i(x)$.

When a read action $r_i(x)$ tries to read x, it must decide which version of x to read. Starting from x_0 which is the original value of x, the $r_i(x)$ will find the version as follows:

Step 1. Find x_j that $j < i$, and there is no other version x_k where $j < k < i$.
Step 2. Determine whether $status(t_j) = aborted$.
- (a) If $status(t_j) \neq aborted$, then x_j is the proper version to read.
- (b) If $status(t_j) = aborted$, then trace back through the chain to find the nearest preceding version x_m where $status(t_m) \neq aborted$.

3.3 Concurrency Control Mechanism for Validation

Rules for Scheduling. The *scheduler* deals with the requests of data actions in MVTO. After the validator constructs all the write chains for data items, scheduler must follow certain rules to make sure the resulting schedule is conflict-serializable and the serial order is identical to the presumptive serial order. The rules are described as follows:

Rule 1. Suppose scheduler receives a request of $w_i(x)$.
- (a) If $w_i(x)$ is legal, *i.e.*, scheduler can find the element of t_i on write chain of x, then scheduler grants the request. Let t_i write x_i into the target element and then set $wb(x_i) = true$.
- (b) If $w_i(x)$ is illegal, then abort t_i, the newly received block fails the validation.

Rule 2. Suppose scheduler receives a request of $r_i(x)$. The scheduler will find out the right version x_j for $r_i(x)$.
- (a) If $j = 0$, grant the request and return the original value of x.
- (b) If $j \neq 0$, then check the status of the read-from transaction t_j.
  - i. If $status(t_j) = Aborted$, then find the proper version again and restart from rule 2.
  - ii. If $status(t_j) = Committed$ then grant the request, return the value of x_j and add $\langle t_j, x_j^i \rangle$ to $rs(t_i)$.
  - iii. If $status(t_j) = Active$
    - A. If $wb(x_j) = true$, grant $r_i(x)$ as in 2(b)ii.
    - B. If $wb(x_j) = false$, delay t_i until $wb(x_j)$ is set to $ture$, then grant $r_i(x)$ as in 2(b)ii.

Rules of Commit. After transaction t_i finishes all its tasks, t_i will try to commit. The rules for committing transaction, which is shown in Algorithm 1, will maintain the recoverability of schedule.

Algorithm 1. Commit t_i

```
1  foreach ⟨t_j, val⟩ ∈ rs(t_i) do
2      while status(t_j) ≠ Committed do
3          if status(t_j) = Aborted then
4              go to: RestartPoint
5          end
6          sleep()
7      end
8      if checkConsistency(⟨t_j, val⟩) = false then
9          go to: RestartPoint
10     end
11 end
12 status(t_i) = Committed
```

As discussed earlier, t_i can't commit until all its read-from transactions (line 1) have committed (line 2) and some of its read-from transactions that are active may abort at any time. Aborting a transaction will discard all its changes to the states and therefore causing the transactions which have read from this aborted transaction have to restart (line 3). When a transaction t_j restarts, $rs(t_j)$ will be cleared and for $\forall x \in ws(t_j)$, $wb(x_j)$ in x's write chain must be set to **false**. The results of the restarted transaction's data actions might change because the read set of transaction will change when its read-from transaction aborts. Thus, when a transaction t_i tries to commit, the original value x_j^i in $rs(t_i)$ may be inconsistent with the newest version x_j because t_j may have restarted and overwritten the original value with new value x_j^*. So t_i must check the consistency between $\forall x_j^i \in rs(t_i)$ and the newest x_j, and t_i must restart when finding the inconsistency (line 8–10). The effect of restart can spread through the write chain after the element of the aborted transaction, making more active transactions restart due to the inconsistency.

Note that the consistency between the committing transaction and its committed read-from transactions will hold because the committed transactions won't restart and change the versions they have written.

4 Correctness

Observation 1. *For conflicting actions $r_i(x)$ and $w_j(x)(i \neq j)$, if $r_i(x)$ read the version x_j which is written by $w_j(x)$, MVTO will ensure $j <= i$ and scheduling rules grant $r_i(x)$ only after $w_j(x)$ finishes.*

Lemma 1. *MVTO will produce conflict-serializable schedule.*

Proof. According to the conflict relation among data actions in final schedule, we can build a precedence-graph [3] which shows the dependency among the

transactions in the block. The conflict notion in MVTO which is presented in Sect. 2.5 shows that the conflict happens between read actions and write action on the same version of the data item. With Observation 1, we can conclude that the precedence-graph is acyclic. As proved elsewhere [3], the schedule with an acyclic precedence graph is conflict-serializable.

Lemma 2. *Final schedule produced by MVTO is conflict-equivalent to serial schedule that matches the transaction order in the block.*

Proof. The precedence-graph of the final schedule produced by MVTO is a directed acyclic graph (Lemma 1). We can observe that the result of the topological sort of this precedence-graph is the serial order of the final schedule. In this graph, for all edges that are pointing from t_i to t_j, MVTO makes sure that $i < j$ (Observation 1), where i and j are also the index of transactions in the block. Therefore, the serial order of schedule produced by MVTO is equivalent to transaction order in the block. Thus we can conclude that the final schedule is conflict-equivalent to serial schedule that matches the transaction order in the block.

Lemma 3. *Validation will succeed if the block is legal.*

Proof. Concurrent schedules of a block's validation in MVTO will produce the same results for each data action as long as the (i) initial states are identical and (ii) these schedules are *conflict-equivalent*.

Blockchain makes sure the honest miners and validators share the same history of blocks when creating or validating the new block. Therefore, the initial states of the new block in validation phase are identical to those initial states in mining phase.

The final schedule of the validation is *conflict-equivalent* to serial schedule that matches the transaction order in the block (Lemma 2). Meanwhile, since the miner adjusts the transaction order in the block to match the serial order of the produced concurrent schedule, the final schedule of the validation and the concurrent schedule produced by miner are both *conflict-equivalent* to the same serial schedule. Therefore these two schedules are *conflict-equivalent* to each other.

Hence, we can conclude that mining and validation can produce the same result as long as the block is legal.

5 Evaluation

MVTO aims to improve the throughput for blocks validation by executing smart contracts in parallel. We use a benchmark for MVTO that vary the workload of a transaction, the percentage of abortion, the percentage of conflict and the size of the thread pool to evaluate this approach.

5.1 Benchmark

The benchmark is the block that only contains the "ProxyBallot" contract which is shown in Fig. 1. Each block contains 200 transactions. In a transaction that calls "ProxyBallot" contract, the workload is the number of the voters which are collected by this transaction. The conflict percentage is defined to be the percentage of transactions that vote to "proposals[0]". Other non-conflicting transactions will vote to different proposals. To control the abort rate of a block, some transactions are set to abort after they finish.

We set up four experiments where we vary the workload, percentage of abortion, percentage of conflict and size of thread pool respectively. (1) Experiment 1 varies the workload for each transaction from 2000 to 20000 voters with 15% data conflict and 10% abort. (2) Experiment 2 varies the percentage of data conflict in the block from 0% to 100% with 20000 workload and 10% abort. (3) Experiment 3 varies the percentage of abortion in the block from 0% to 100% with 20000 workload and 15% data conflict. All the above three experiments use a thread pool with 3 worker threads. (4) Experiment 4 varies the size of the thread pool from 3 to 15 and the conflict percentage from 5% to 15% with 20000 workload and 10% abort.

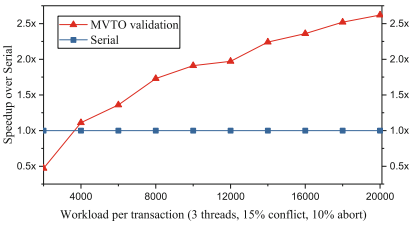
5.2 Results

We use `C++` implementation to run the evaluation on a machine with 4-core 4.00 GHz CPU. All the results are the mean of 100 times executions. Results are shown in Fig. 4, each contains the results of serial and concurrent validation on the same block. Results of serial validation serve as the baseline when showing MVTO's speedup.

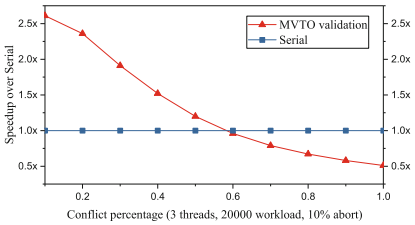
Figure 4a shows the speedup of MVTO over serial validation when varying the workload per transaction with 15% conflict and 10% abort. Because of the overhead of multithreading, MVTO is slower than serial validation when workload is lower than 4000. MVTO achieves speedup when workload is higher than 4000 and achieves 2.5x speedup when workload is 20000.

Figure 4b shows the result when varying the conflict percentage of the block with 20000 workload per transaction and 10% abort. As the conflict percentage raise, the speedup of MVTO keeps dropping from 2.5x to nearly 0.5x. When conflict percentage reaches 60%, MVTO becomes slower than serial validation.

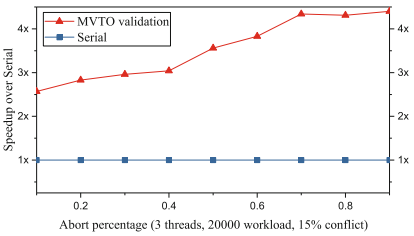
Figure 4c shows the speedup of MVTO when varying the abort percentage of the block with 20000 workload per transaction and 15% conflict. We can observe that serial execution is significantly slower when there are more transactions abort during validation. The reason is the difference between single-version and multiversion when dealing with transaction's abortion. To abort transactions in single-version, the undo logs are commonly needed to roll back every tentative change that made on state variables. While MVTO can just discard the version which is written by the aborted transaction. Hence MVTO shows lower cost of restarting transactions compared to single-version technique.



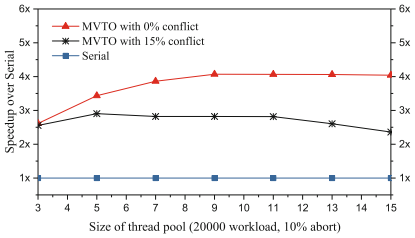
(a) speedup against workload per transaction



(b) speedup against conflict percentage



(c) speedup against abort percentage



(d) speedup against size of thread pool

Fig. 4. Evaluation results

Figure 4d shows that how the size of thread pool affect the speedup of MVTO under blocks with or without conflict. When the block admit no conflict, the speedup is generally higher with lager thread pool and MVTO cannot achieve more than 4x speedup due to the limitation of 4-core CPU. However, when the block contains 15% conflict, the effect of speedup begins to drop when using thread pool larger than 5 threads. This is because more active threads can lead to more possibility to have synchronization on the conflicting data item between active transactions at runtime.

5.3 Discussion

As we mentioned in Sect. 1, Ethereum can deal with more complicated and time-consuming contracts when it switch to POS. Hence, despite MVTO can only achieve desirable speedup when executing smart contracts with enough workload due to the overhead of multithreading (Fig. 4a), it can bring speedup to Ethereum and any other platforms which will deal with complicated smart contracts.

The results in Fig. 4c shows the advantage of multiversion technique used in MVTO compared to single-version implementation when dealing with data roll back which caused by aborting and restarting transactions. Results in Fig. 4d shows that MVTO may slow down when using a thread pool with too much worker threads because of the overhead of the synchronization between conflicting data action. In this evaluation, the transactions in benchmark only calls the

same smart contract so that the conflict percentage can be very high (as shown in Fig. 4b). In practice, miner will receive many transactions that trigger unrelated smart contracts, thus the blocks in reality might face less data conflict than the blocks used in this evaluation.

6 Related Work

Dickerson *et al.* [4] treat every smart contract invocation as a speculative atomic action. Therefore the miner can discover a serializable schedule when creating the new block by using locks in a *2PL* manner. When executing, the miner records the schedule by storing the trace of the lock on every data item into the block, so that validator can retrieve and replay that same schedule deterministically. Serge and Hobor [5] present the similarities between multi-transactional behaviors of smart contract and problems of shared-memory concurrency. Bocchino *et al.* [6] survey many techniques for replaying a concurrent schedule deterministically.

Ethereum [1] may be the most popular platform among all the smart contract platforms on public blockchain. Its recent project Plasma [7] tries to remission the low throughput by employing the sharding techniques. Plasma split the state space into multiple partitions where each runs on a different child blockchain, forming a multiple blockchains ecosystem with tree hierarchy. Other ongoing platforms such as Polkadot [8], EOS [9] and Aelf [10] also adopt similar techniques to parallelize contracts that working on different state space.

Garcia-Molina *et al.* [3] introduces concurrency control mechanism that employ multiversion techniques. Many software transactional memory (STM) techniques [11,12] fit well with smart contract because smart contracts must be executed transactionally. Ghosh *et al.* [13] uses "update chain" that similar to the write chain in MVTO to implement a concurrency control mechanism using multiversion timestamp under STM. This mechanism aims to reduce aborts when facing conflicting update transactions. The "update chain" is constructed at runtime while MVTO constructs write chain before the execution.

7 Conclusion

We have proposed a concurrent scheme called MVTO to increase the throughput of blockchain-based smart contract platform. In this scheme, the miner can use any concurrency control technique to discover a conflict-serializable schedule. The write set of each transaction is recorded into the newly generated block along with the final states. The order of transactions in block is adjusted by the miner to match the serial order of the resulting schedule thereby validators can know which serial order they should replay. Before the validation, validators use those write sets and the transaction order to construct "write chain" on conflicting data items, therefore they can pre-determine the dependency among transactions in the block. In summary, validators can validate a block in a concurrent and deterministic manner. The Evaluation shows that MVTO can achieve

approximately 2.5x speedup when validating a block with conflicting input data using a thread pool with 3 worker threads.

Furthermore, MVTO can be integrated into existing systems without compromising their original architecture. Developers just need to implement the concurrency control and multiversion on the underlying basic data items. Existing platforms mostly plan to increase throughput by employing sharding. MVTO is compatible with the sharding techniques where sharding scale through the multi-chains architecture and MVTO enable concurrency within each child chain. Hence, MVTO can achieve significant scalability by using sharding techniques to divide the states into different partitions.

In conclusion, MVTO can speed up the block validation and increase the throughput of smart contract platforms.

References

1. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, vol. 151 (2014)
2. Ethereum: Solidity documentation. http://solidity.readthedocs.io/en/develop
3. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database System Implementation, pp. 883–940. Prentice-Hall, Upper Saddle River (2000)
4. Dickerson, T.D., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, 25–27 July 2017, pp. 303–312 (2017)
5. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: FC 2017 International Workshops on Financial Cryptography and Data Security, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, 7 April 2017, pp. 478–493 (2017). Revised Selected Papers
6. Bocchino, R., Adve, V., Adve, S., Snir, M.: Parallel programming must be deterministic by default. In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, p. 4 (2009)
7. Poon, J., Buterin, V.: Plasma: scalable autonomous smart contracts. White Paper (2017)
8. Wood, G.: Polkadot: vision for a heterogeneous multi-chain framework. White Paper (2016)
9. EOS: EOS.IO technical white paper. https://eos.io/
10. AELF: a multi-chain parallel computing blockchain framework. https://aelf.io/
11. Herlihy, M., Luchangco, V., Moir, M., Scherer III., W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, 13–16 July 2003, pp. 92–101 (2003)
12. Zhang, D., Dechev, D.: Lock-free transactions without rollbacks for linked data structures. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, 11–13 July 2016, pp. 325–336 (2016)
13. Ghosh, A., Chaki, R., Chaki, N.: A new concurrency control mechanism for multithreaded environment using transactional memory. J. Supercomput. **71**(11), 4095–4115 (2015)