An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts*†

Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani
Department of Computer Science and Engineering, IIT Hyderabad, India
{cs17resch11004, cs15resch01004, sathya_p, cs18mtech01002, cs15resch01001}@iith.ac.in

***Abstract*—Blockchain platforms such as Ethereum and several others execute *complex transactions* in blocks through user-defined scripts known as *smart contracts*. Normally, a block of the chain consists of multiple transactions of smart contracts which are added by a *miner*. To append a correct block into the blockchain, miners execute these transactions of smart contracts sequentially. Later the *validators* serially re-execute the smart contract transactions of the block. If the validators agree with the final state of the block as recorded by the miner, then the block is said to be validated. It is then added to the blockchain using a consensus protocol. In Ethereum and other blockchains that support cryptocurrencies, a miner gets an incentive every time such a valid block successfully added to the blockchain.**

In most of the current day blockchains the miners and validators execute the smart contract transactions serially. In the current era of multi-core processors, by employing the serial execution of the transactions, the miners and validators fail to utilize the cores properly and as a result, have poor throughput. By adding concurrency to smart contracts execution, we can achieve better efficiency and higher throughput. In this paper, we develop an efficient framework to execute the smart contract transactions concurrently using optimistic *Software Transactional Memory systems* (STMs).

Miners execute smart contract transactions concurrently using multi-threading to generate the final state of blockchain. STM is used to take care of synchronization issues among the transactions and ensure atomicity. Now when the validators also execute the transactions (as a part of validation) concurrently using multi-threading, then the validators may get a different final state depending on the order of execution of conflicting transactions. To avoid this, the miners also generate a block graph of the transactions during the concurrent execution and store it in the block. This graph captures the conflict relations among the transactions and is generated concurrently as the transactions are executed by different threads.

The miner proposes a block which consists of set of transactions, block graph, hash of the previous block, and final state of each shared data-objects. Later, the validators re-execute the same smart contract transactions concurrently and deterministically with the help of block graph given by the miner to verify the final state. If the validation is successful then proposed block appended into the blockchain and miner gets incentive otherwise discard the proposed block.

We execute the smart contract transactions concurrently using Basic Time stamp Ordering (BTO) and Multi-Version Time stamp Ordering (MVTO) protocols as optimistic STMs. BTO and MVTO miner achieves 3.6x and 3.7x average speedups over serial miner respectively. Along with, BTO and MVTO validator outperform average 40.8x and 47.1x than serial validator respectively.

*Author sequence follows lexical order of last names.
†The proposal of this paper has been accepted in Doctoral Symposium, ICDCN'19.

***Index Terms*—Blockchain, Smart Contracts, Software Transactional Memory System, Multi-version Concurrency Control, Opacity**

I. Introduction

It is commonly believed that blockchain is a revolutionary technology for doing business over the Internet. Blockchain is a decentralized, distributed database or ledger of records. Cryptocurrencies such as Bitcoin [15] and Ethereum [5] were the first to popularize the blockchain technology. Blockchains ensure that the records are tamper-proof but publicly readable.

Basically, the blockchain network consists of multiple peers (or nodes) where the peers do not necessarily trust each other. Each node maintains a copy of the distributed ledger. *Clients*, users of the blockchain, send requests or *transactions* to the nodes of the blockchain called as *miners*. The miners collect multiple transactions from the clients and form a *block*. Miners then propose these blocks to be added to the blockchain. They follow a global consensus protocol to agree on which blocks are chosen to be added and in what order. While adding a block to the blockchain, the miner incorporates the hash of the previous block into the current block. This makes it difficult to tamper with the distributed ledger. The resulting structure is in the form of a linked list or a chain of blocks and hence the name blockchain.

The transactions sent by clients to miners are part of a larger code called as *smart contracts* that provide several complex services such as managing the system state, ensuring rules, or credentials checking of the parties involved [3]. Smart contracts are like a 'class' in programming languages that encapsulate data and methods which operate on the data. The data represents the state of the smart contract (as well as the blockchain) and the methods (or functions) are the transactions that possibly can change contract state. A transaction invoked by a client is typically such a method or a collection of methods of the smart contracts. Ethereum uses Solidity [4] while Hyperledger supports language such as Java, Golang, Node.js etc.

Motivation for Concurrent Execution of Smart Contracts: As observed by Dickerson et al. [3], smart contract transactions are executed in two different contexts specifically in Ethereum. First, they are executed by miners while forming a block. A miner selects a sequence of client request transactions, executes the smart contract code of these transactions in sequence, transforming the state of the associated contract in this process. The miner then stores the sequence of transactions, the resulting

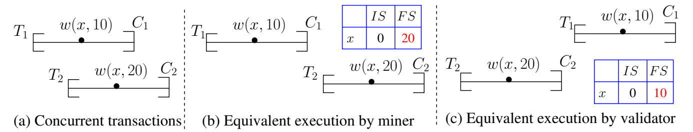
final state of the contracts in the block along with the hash of the previous block. After creating the block, the miner proposes it to be added to the blockchain through the consensus protocol.

Once a block is added, the other peers in the system, referred to as *validators* in this context, validate the contents of the block. They re-execute the smart contract transactions in the block to verify the block's final states match or not. If final states match, then the block is accepted as valid and the miner who appended this block is rewarded. Otherwise, the block is discarded. Thus the transactions are executed by every peer in the system. In this setting, it turns out that the validation code runs several times more than miner code [3].

This design of smart contract execution is not very efficient as it does not allow any concurrency. Both the miner and the validator execute transactions serially one after another. In today's world of multi-core systems, the serial execution does not utilize all the cores and hence results in lower throughput. This limitation is not specific only to Ethereum but almost all the popular blockchains. Higher throughput means more number of transactions executed per unit time by miners and validators which clearly will be desired by both of them.

But the concurrent execution of smart contract transactions is not an easy task. The various transactions requested by the clients could consist of conflicting access to the shared data-objects. Arbitrary execution of these transactions by the miners might result in the data-races leading to the inconsistent final state of the blockchain. Unfortunately, it is not possible to statically identify if two contract transactions are conflicting or not since they are developed in Turing-complete languages. The common solution for correct execution of concurrent transactions is to ensure that the execution is *serializable* [16]. A usual correctness-criterion in databases, serializability ensure that the concurrent execution is equivalent to some serial execution of the same transactions. Thus the miners must ensure that their execution is serializable [3] or one of its variants as described later.

The concurrent execution of the smart contract transactions of a block by the validators although highly desirable can further complicate the situation. Suppose a miner ensures that the concurrent execution of the transactions in a block are serializable. Later a validator executes the same transactions concurrently. But during the concurrent execution, the validator may execute two conflicting transactions in an order different from what was executed by the miner. Thus the serialization order of the miner is different from the validator. Then this can result in the validator obtaining a final state different from what was obtained by the miner. Consequently, the validator may incorrectly reject the block although it is valid. Figure 1 illustrates this in the following example. Figure 1 (a) consists of two concurrent conflicting transactions T_1 and T_2 working on same shared data-objects x which are part of a block. Figure 1 (b) represents the concurrent execution by miner with an equivalent serial schedule as T_1, T_2 and final state (or FS) as 20 from the initial state (or IS) 0. Whereas Figure 1 (c), shows the concurrent execution by a validator with an equivalent serial schedule as T_2, T_1, and final state as 10 from IS 0 which



(a) Concurrent transactions (b) Equivalent execution by miner (c) Equivalent execution by validator

Fig. 1. Execution of concurrent transactions by miner and validator

is different from the final state proposed by the miner. Such a situation leads to rejection of the valid block by the validator which is undesirable. These important issues were identified by Dickerson et al. [3] who proposed a solution of concurrent execution for both the miners and validators. In their solution, the miners concurrently execute the transactions of a block using abstract locks and inverse logs to generate a serializable execution. Then, to enable correct concurrent execution by the validators, the miners also provide a *happens-before* graph in the block. The happens-before graph is a direct acyclic graph over all the transaction of the block. If there is a path from a transaction T_i to T_j then the validator has to execute T_i before T_j. Transactions with no path between them can execute concurrently. The validator using the happens-before graph in the block executes all the transactions concurrently using the fork-join approach. This methodology ensures that the final state of the blockchain generated by the miners and the validators are the same for a valid block and hence not rejected by the validators. The presence of tools such as a happens-before graph in the block provides greater enhancement to validators to consider such blocks as it helps them execute quickly by means of parallelization as opposed to a block which does not have any tools for parallelization. This, in turn, entices the miners to provide such tools in the block for concurrent execution by the validators.

Our Solution Approach - Optimistic Concurrent Execution and Lock-Free Graph: Dickerson et al. [3] developed a solution to the problem of concurrent miner and validators using locks and inverse logs. It is well known that locks are pessimistic in nature. So, in this paper, we explore a novel and efficient framework for concurrent miners using optimistic Software Transactional Memory Systems (STMs).

The requirement of the miner, as explained above, is to concurrently execute the smart contract transactions correctly and output a graph capturing dependencies among the transactions of the block such as happens-before graph. We denote this graph as *block graph* (or BG). In the proposed solution, the miner uses the services of an optimistic STM system to concurrently execute the smart contract transactions. Since STMs also work with transactions, we differentiate between smart contract transactions and STM transactions. The STM transactions invoked by an STM system is a piece of code that it tries to execute atomically even in presence of other concurrent STM transactions. If the STM system is not able to execute it atomically, then the STM transaction is aborted.

The expectation of a smart contract transaction is that it will be executed serially. Thus, when it is executed in a concurrent setting, it is expected to be executed atomically (or serialized). But unlike STM transaction, a smart contract transaction cannot

be committed or aborted. Thus to differentiate between smart contract transaction from STM transaction, we denote smart contract transaction as *Atomic Unit* or *atomic-unit* and STM transaction as transaction in the rest of the document. Thus the miner uses the STM system to invoke a transaction for each atomic-unit. In case the transaction gets aborted, then the STM repeatedly invokes new transactions for the same atomic-unit until a transaction invocation eventually commits.

A popular correctness guarantee provided by STM systems is *opacity* [6] which is stronger than serializability. Opacity like serializability requires that the concurrent execution including the aborted transactions be equivalent to some serial execution. This ensures that even aborted transaction reads consistent value until the point of abort. As a result, that the application such as a miner using an STM does not encounter any undesirable side-effects such as crash failures, infinite loops, divide by zero, etc. STMs provide this guarantee by executing optimistically and support atomic (opaque) reads, writes on *transactional objects* (or t-object*s*).

Among the various STMs available, we have chosen two timestamp based STMs in our design: (1) *Basic Timestamp Ordering* or *BTO* STM [20, Chap 4], maintains only one version for each *t-object*. (2) *Multi-Version Timestamp Ordering* or *MVTO* STM [11], maintains multiple versions corresponding to each *t-object* which further reduces the number of aborts and improves the throughput.

The advantage of using timestamp based STM is that in these systems the equivalent serial history is ordered based on the timestamps of the transactions. Thus using the timestamps, the miner can generate the BG of the atomic-units. Dickerson et al. [3], developed the BG in a serial manner. In our approach, the graph is developed by the miner in concurrent and lock-free [9] manner.

The validator process creates multiple threads. Each of these threads parses the BG and re-execute the atomic-units for validation. The BG provided by concurrent miner shows dependency among the atomic-units. Each validator thread, claims a node which does not have any dependency, i.e. a node without any incoming edges by marking it. After that, it executes the corresponding atomic-units deterministically. Since the threads execute only those nodes that do not have any incoming edges, the concurrently executing atomic-units will not have any conflicts. Hence the validator threads need not have to worry about synchronization issues. We denote this approach adopted by the validator as a decentralized approach (or Decentralized Validator) as the multiple threads are working on BG concurrently in the absence of master thread.

The approach adopted by Dickerson et al. [3], works on fork-join in which a master thread allocates different tasks to slave threads. The master thread will identify those atomic-units which do not have any dependencies from the BG and allocates them to different slave threads to work on. In this paper, we compare the performance of both these approaches with the serial validator.

Contributions of the paper as follows:

- Introduce a novel way to execute the smart contract transactions by concurrent miner using optimistic STMs.
- We implement the concurrent miner with the help of BTO and MVTO STM but its generic to any STM protocol.
- We propose a lock-free graph library to generate the BG.
- We propose concurrent validator that re-executes the smart contract transactions deterministically and efficiently with the help of BG given by concurrent miner.
- Concurrent miner satisfies correctness criterion as opacity.
- We achieve 3.6x and 3.7x average speedups for concurrent miner using BTO and MVTO STM protocol respectively. Along with, BTO and MVTO validator outperform average 40.8x and 47.1x than serial validator respectively.

Related Work: The first *blockchain* concept has been given by Satoshi Nakamoto in 2009 [15]. He proposed a system as bitcoin [15] which performs electronic transactions without the involvement of the third party. The term smart contract [19] has been introduced by Nick Szabo. *Smart contract* is an interface to reduce the computational transaction cost and provides secure relationships on public networks. There exist few paper in the literature that works on safety and security concern of smart contracts. Luu et al. [14] addresses the waste part of the computational effort by miner that can be utilized and lead to award the incentives. Delmolino et al. [2] document presents the common pitfall made while designing a secure smart contract. Nowadays, ethereum [5] is one of the most popular smart contract platform which supports a built-in Turing-complete programming language. Ethereum virtual machine (EVM) uses Solidity [4] programming language. Luu et al. [13] addresses several security problems and proposed an enhanced mechanism to make the ethereum smart contracts less vulnerable.

Sergey et al. [17] elaborates a new perspective between smart contracts and concurrent objects. Zang et al. [21] uses any concurrency control mechanism for concurrent miner which delays the read until the corresponding writes to commit and ensures conflict-serializable schedule. Basically, they proposed concurrent validators using MVTO protocol with the help of write sets provided by the concurrent miner. Dickerson et al. [3] introduces a speculative way to execute smart contracts by using concurrent miner and concurrent validators. They have used pessimistic software transactional memory systems (STMs) to execute concurrent smart contracts which use rollback if any inconsistency occurs and prove that schedule generated by concurrent miner is *serializable*. We proposed an efficient framework for concurrent execution of the smart contracts using optimistic software transactional memory systems. So, the updates made by a transaction will be visible to shared memory only on commit hence, rollback is not required. Our approach ensures correctness criteria as opacity [6] which considers aborted transactions as well because it read correct values.

Weikum et al. [20] proposed concurrency control techniques that maintain single-version and multiple versions corresponding to each data-object. STMs [8], [18] are alternative to locks for addressing synchronization and concurrency issues in multi-core systems. STMs are suitable for the concurrent executions

of smart contracts without worrying about consistency issues. Single-version STMs protocol store single version corresponding to each data-object as BTO STM. It identifies the conflicts between two transactions at run-time and abort one of them and retry again for the aborted transaction. Kumar et al. [11] observe that storing multiple versions corresponding to each data-object reduces the number of aborts and provides greater concurrency that leads to improving the throughput.

II. System Model and Background

This section includes the commencement of notions related to this paper such as blockchain, smart contracts, STMs and its execution model. Here, we limit our discussion to a well-known smart contracts platform, Ethereum. We improve the throughput by ensuring the concurrent execution of smart contracts using an efficient framework, optimistic STMs.

A. Blockchain and Smart Contracts

Blockchain is a distributed and highly secure technology which stores the records into the block. It consists of multiple peers (or nodes), and each peer maintains decentralize distributed ledger that makes it publicly readable but tamper-proof. Peer executes some functions in the form of transactions. A transaction is a set of instructions executing in the memory. Bitcoin is a blockchain system which only maintains the balances while transferring the money from one account to another account in the distributed manner. Whereas, the popular blockchain system such as Ethereum maintains the state information as well. Here, transactions execute the atomic code known as a function of smart contract. Smart contract consists of one or more atomic-units or functions. In this paper, the atomic-unit contains multiple steps that have been executed by an efficient framework which is optimistic STMs.

Smart Contracts: The transactions sent by clients to miners are part of a larger code called as *smart contracts* that provide several complex services such as managing the system state, ensures rules, or credentials checking of the parties involved, etc. [3]. For better understanding of smart contract, we describe a simple auction contract from Solidity documentation [4].

Simple Auction Contract: The functionality of simple auction contract is shown in Algorithm 1. Where Line 1 declares the contract, followed by public state variables as "highestBidder, highestBid, and pendingReturn" which records the state of the contract. A single owner of the contract initiates the auction by executing constructor "SimpleAuction()" method (omitted due to lack of space) in which function initialize bidding time as auctionEnd (Line 3). There can be any number of participants to bid. The bidders may get their money back whenever the highest bid is raised. For this, a public state variable declared at Line 7 (pendingReturns) uses solidity built-in complex data type mapping to maps bidder addresses with unsigned integers (withdraw amount respective to bidder). Mapping can be seen as a hash table with key-value pair. This mapping uniquely identifies account addresses of the clients in the Ethereum blockchain. A bidder withdraws the amount of their earlier bid by calling withdraw() method [4].

At Line 8, a contract function "bid()" is declared, which is called by bidders to bid in the auction. Next, "auctionEnd" variable is checked to identify whether the auction already called off or not. Further, bidders "msg.value" check to identify the highest bid value at Line 12. Smart contract methods can be aborted at any time via throw when the auction is called off, or bid value is smaller than current "highestBid". When execution reaches to Line 16, the "bid()" method recovers the current highest bidder data from mapping through the "highestBidder" address and updates the current bidder pending return amount. Finally, at Line 18 and Line 19, it updates the new highest bidder and highest bid amount.

Algorithm 1 SimpleAuction: It allows every bidder to send their bids throughout the bidding period.

```
1:  procedure CONTRACT SIMPLEAUCTION
2:      address public beneficiary;
3:      uint public auctionEnd;
4:      /*current state of the auction*/
5:      address public highestBidder;
6:      uint public highestBid;
7:      mapping(address => uint) pendingReturns;
8:      function bid() public payable
9:          if (now ≥ auctionEnd) then
10:             throw;
11:         end if
12:         if (msg.value < highestBid) then
13:             thorw;
14:         end if
15:         if (highestBid != 0) then
16:             pendingReturns[highestBidder] += highestBid;
17:         end if
18:         highestBidder = msg.sender;
19:         highestBid = msg.value;
20:     end function
21:     // more operation definitions
22: end procedure
```

Software Transactional Memory Systems: Following [7], [12], we assume a system of n processes/threads, $p_1, \ldots, p_n$ that access a collection of *transactional objects* or *t-objects* via atomic *transactions*. Each transaction has a unique identifier. Within a transaction, processes can perform *transactional operations or methods*: *STM.begin()* that begins a transaction, *STM.write*(x, v) (or $w(x, v)$) that updates a *t-object* x with value v in its local memory, *STM.read*(x, v) (or $r(x, v)$) that tries to read x and returns value as v, *STM.tryC*$()$ that tries to commit the transaction and returns $commit$ (or $\mathcal{C}$) if it succeeds, and *STM.tryA*$()$ that aborts the transaction and returns $\mathcal{A}$. Operations *STM.read()* and *STM.tryC*$()$ may return $\mathcal{A}$. Transaction T_i starts with the first operation and completes when any of its operations return $\mathcal{A}$ or $\mathcal{C}$. For a transaction T_k, we denote all the *t-object*s accessed by its read operations and write operations as $rset_k$ and $wset_k$ respectively. We denote all the operations of a transaction T_k as $evts(T_k)$ or $evts_k$.

History: A *history* is a sequence of *events*, i.e., a sequence of invocations and responses of transactional operations. The collection of events is denoted as $evts(H)$. For simplicity, we only consider *sequential* histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event and let $<_H$ denote the total order on the transactional operations incurred by H. We identify a history

H as tuple $\langle evts(H), <_H \rangle$.

We only consider *well-formed* histories here, i.e., no transaction of a process begins before the previous transaction invocation has completed (either $commits$ or $aborts$). We also assume that every history has an initial *committed* transaction T_0 that initializes all the t-objects with value 0. The set of transactions that appear in H is denoted by *txns*(H). The set of *committed* (resp., *aborted*) transactions in H is denoted by *committed(H)* (resp., *aborted*(H)). The set of *incomplete* or *live* transactions in H is denoted by $H.incomp = H.live = (txns(H) - committed(H) - aborted(H))$.

We construct a *complete history* of H, denoted as $\overline{H}$, by inserting $STM.tryA_k(\mathcal{A})$ immediately after the last event of every transaction $T_k \in H.live$. But for $STM.tryC_i$ of transaction T_i, if it released the lock on first *t-object* successfully that means updates made by T_i is consistent so, T_i will immediately return commit.

Transaction Real-Time and Conflict order: For two transactions $T_k, T_m \in txns(H)$, we say that T_k *precedes* T_m in the *real-time order* of H, denoted as $T_k \prec_H^{RT} T_m$, if T_k is complete in H and the last event of T_k precedes the first event of T_m in H. If neither $T_k \prec_H^{RT} T_m$ nor $T_m \prec_H^{RT} T_k$, then T_k and T_m *overlap* in H. We say that a history is *serial* (or *t-sequential*) if all the transactions are ordered by real-time order.

We say that T_k, T_m are in conflict, denoted as $T_k \prec_H^{Conf} T_m$, if (1) $STM.tryC_k() <_H STM.tryC_m()$ and $wset(T_k) \cap wset(T_m) \neq \emptyset$; (2) $STM.tryC_k() <_H r_m(x,v)$, $x \in wset(T_k)$ and $v \neq \mathcal{A}$; (3) $r_k(x,v) <_H STM.tryC_m()$, $x \in wset(T_m)$ and $v \neq \mathcal{A}$. Thus, it can be seen that the conflict order is defined only on operations that have successfully executed. We denote the corresponding operations as conflicting.

Valid and Legal histories: A successful read $r_k(x,v)$ (i.e., $v \neq \mathcal{A}$) in a history H is said to be *valid* if there exist a transaction T_j that wrote v to x and *committed* before $r_k(x,v)$. History H is valid if all its successful read operations are valid.

We define $r_k(x,v)$'s *lastWrite* as the latest commit event $\mathcal{C}_i$ preceding $r_k(x,v)$ in H such that $x \in wset_i$ (T_i can also be T_0). A successful read operation $r_k(x,v)$ (i.e., $v \neq \mathcal{A}$), is said to be *legal* if the transaction containing r_k's lastWrite also writes v onto x. The history H is legal if all its successful read operations are legal. From the definitions we get that if H is legal then it is also valid.

Notions of Equivalence: Two histories H and H' are *equivalent* if they have the same set of events. We say two histories H, H' are *multi-version view equivalent* [20, Chap. 5] or *MVVE* if (1) H, H' are valid histories and (2) H is equivalent to H'.

Two histories H, H' are *view equivalent* [20, Chap. 3] or *VE* if (1) H, H' are legal histories and (2) H is equivalent to H'. By restricting to legal histories, view equivalence does not use multi-versions.

Two histories H, H' are *conflict equivalent* [20, Chap. 3] or *CE* if (1) H, H' are legal histories and (2) conflict in H, H' are the same, i.e., $conf(H) = conf(H')$. Conflict equivalence like view equivalence does not use multi-versions and restricts itself to legal histories.

VSR, MVSR, and CSR: A history H is said to VSR (or View Serializable) [20, Chap. 3], if there exist a serial history S such that S is view equivalent to H. But this notion considers only single version corresponding to each *t-object*.

MVSR (or Multi-Version View Serializable) maintains multiple version corresponding to each *t-object*. A history H is said to MVSR [20, Chap. 5], if there exist a serial history S such that S is multi-version view equivalent to H. It can be proved that verifying the membership of VSR as well as MVSR in databases is NP-Complete [16]. To circumvent this issue, researchers in databases have identified an efficient subclass of VSR, called CSR based on the notion of conflicts. The membership of CSR can be verified in polynomial time using conflict graph characterization.

A history H is said to CSR (or Conflict Serializable) [20, Chap. 3], if there exist a serial history S such that S is conflict equivalent to H.

Serializability and Opacity: Serializability [16] is a commonly used criterion in databases. But it is not suitable for STMs as it does not consider the correctness of *aborted* transactions as shown by Guerraoui and Kapalka [6]. Opacity, on the other hand, considers the correctness of *aborted* transactions as well.

A history H is said to be *opaque* [6], [7] if it is valid and there exists a t-sequential legal history S such that (1) S is equivalent to complete history $\overline{H}$ and (2) S respects $\prec_H^{RT}$, i.e., $\prec_H^{RT} \subset \prec_S^{RT}$. By requiring S being equivalent to $\overline{H}$, opacity treats all the incomplete transactions as aborted.

Linearizability: A history H is linearizable [10] if (1) The invocation and response events can be reordered to get a valid sequential history. (2) The generated sequential history satisfies the objects sequential specification. (3) If a response event precedes an invocation event in the original history, then this should be preserved in the sequential reordering.

Lock Freedom: An algorithm is said to be lock-free [9] if the program threads are run for a sufficiently long time, at least one of the threads makes progress. It allows individual threads to starve but guarantees system-wide throughput.

III. Requirements of Concurrent Miner, Validator and Block Graph

This section describes the requirements of concurrent miner, validator and block graph to ensure correct concurrent execution of the smart contract transactions.

A. Requirements of the Concurrent Miner

The miner process invokes several threads to concurrently execute the smart contract transactions or atomic-units. With the proposed optimistic execution approach, each miner thread invokes an atomic-unit as a transaction.

The miner should ensure the correct concurrent execution of the smart contract transactions. The incorrect concurrent execution (or consistency issues) may occur when concurrency involved. Any inconsistent read may leads system to divide

by zero, infinite loops, crash failure etc. All smart contract transactions take place within a virtual machine [3]. When miner executes the smart contract transactions concurrently on the virtual machine then infinite loop and inconsistent read may occur. So, to ensure the correct concurrent execution, the miner should satisfy the correctness-criterion as opacity [6].

To achieve better efficiency, sometimes we need to adapt the non-virtual machine environment which necessitates with the safeguard of transactions. As well miner needs to satisfies the correctness-criterion as opacity to ensure the correct concurrent execution of smart contract transactions.

Requirement 1: Any history H_m generated by concurrent miner should satisfy opacity.

Concurrent miner maintains a BG and provides it to concurrent validators which ensures the dependency order among the conflicting transactions. As we discussed in Section I, if concurrent miner will not maintain the BG then a valid block may get rejected by the concurrent validator.

B. Requirements of the Concurrent Validator

The correct concurrent execution by validator should be equivalent to some serial execution. The serial order can be obtained by applying the topological sort on the BG provided by the concurrent miner. BG gives partial order among the transactions while restricting the dependency order same as the concurrent miner. So validator executes those transactions concurrently which are not having any dependency among them with the help of BG. Validator need not have to worry about any concurrency control issues because BG ensures conflicting transactions never execute concurrently.

C. Requirements of the Block Graph

As explained above, the miner generates a BG to capture the dependencies between the smart contract transactions which is used by the validator to concurrently execute the transactions again later. The validator executes those transactions concurrently which do not have any path (implying dependency) between them. Thus the execution by the validator is given by a topological sort on the BG.

Now it is imperative that the execution history generated by the validator, H_v is 'equivalent' to the history generated by the miner, H_m. The precise equivalence depends on the STM protocol followed by the miners and validators. If the miner uses Multi-version STM such as MVTO then the equivalence between H_v and H_m is MVVE. In this case, the graph generated by the miner would be multi-version serialization graph [20, Chap. 5].

On the other hand, if the miner uses single version STM such as BTO then the equivalence between H_v and H_m is view-equivalence (VE) which can be approximated by conflict-equivalence (CE). Hence, in this case, the graph generated by the miner would be conflict graph [20, Chap. 3].

IV. Proposed Mechanism

This section presents the methods of lock-free concurrent block graph library followed by concurrent execution of smart contract transactions by miner and validator.

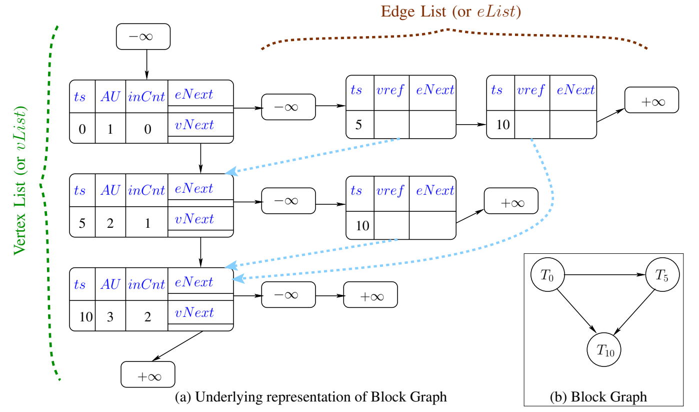


Fig. 2. Pictorial representation of Block Graph

A. Lock-free Concurrent Block Graph

Data Structure of Lock-free Concurrent Block Graph: We use *adjacency list* to maintain the block graph BG(V, E) as shown in Figure 2 (a). Where V is set of vertices (or *vNode*s) which are stored in the vertex list (or *vList*) in increasing order of timestamp between two sentinel node *vHead* (-∞) and *vTail* (+∞). Each vertex node (or *vNode*) contains $\langle ts = i, AU_{id} = id, inCnt = 0, vNext = nil, eNext = nil\rangle$. Where i is a unique timestamp (or ts) of committed transactions T_i. AU_{id} is the id of atomic-unit which is executed by transaction T_i. To maintain the indegree count of each *vNode* we initialize *inCnt* as 0. *vNext* and *eNext* initializes as nil.

Here, E is a set of edges which maintains all the conflicts of *vNode* in the edge list (or *eList*) as shown in Figure 2 (a). *eList* stores *eNode*s (or conflicting transaction nodes say T_j) in increasing order of timestamp (or ts) between two sentinel nodes *eHead* (-∞) and *eTail* (+∞).

Edge node (or *eNode*) contains $\langle$*ts = j, vref, eNext = nil* $\rangle$. Here, j is a unique timestamp (or ts) of *committed* transaction T_j which is having conflict with T_i and $ts(T_i)$ is less than $ts(T_j)$. To maintain the acyclicity of the BG, we add a conflict edge from lower timestamp transaction to higher timestamp transaction i.e. conflict edge is from T_i to T_j in the BG. Figure 2 (b) illustrates this using three transactions with timestamp 0, 5, and 10, which maintain the acyclicity while adding an edge from lower to higher timestamp. *Vertex node reference (or vref)* keeps the reference of its own vertex which is present in the *vList*. *eNext* initializes as nil.

Block graph generated by the concurrent miner which helps to execute the validator concurrently and deterministically through lock-free graph library methods. Lock-free graph library consists of five methods as follows: *addVert(), addEdge(), searchLocal(), searchGlobal()* and *decInCount()*.

Lock-free Graph Library Methods Accessed by Concurrent Miner: Concurrent miner uses addVert() and addEdge() methods of lock-free graph library to build a BG. When concurrent miner wants to add a node in the BG then first it calls addVert() method. addVert() method identifies the correct location of that node (or *vNode*) in the *vList*. If *vNode* is not

part of *vList* then it creates the node and adds it into *vList* in lock-free manner with the help of atomic compare and swap operation.

After successful addition of *vNode* in the BG concurrent miner calls addEdge() method to add the conflicting node (or *eNode*) corresponding to *vNode* in the *eList*. First, addEdge() method identifies the correct location of *eNode* in the *eList* of corresponding *vNode*. If *eNode* is not part of *eList* then it creates the node and adds it into *eList* of *vNode* in lock-free manner with the help of atomic compare and swap operation. After successful addition of *eNode* in the *eList* of *vNode*, it increment the *inCnt* of *eNode*.$vref$ (to maintain the indegree count) node which is present in the *vList*.

Lock-free Graph Library Methods Accessed by Concurrent Validator: Concurrent validator uses searchLocal(), searchGlobal() and decInCount() methods of lock-free graph library. First, concurrent validator thread calls searchLocal() method to identify the source node (having indegree (or *inCnt*) 0) in its local *cacheList* (or thread local memory). If any source node exist in the local *cacheList* with *inCnt* 0 then it sets *inCnt* field to be -1 atomically to claim the ownership of the node.

If the source node does not exists in the local *cacheList* then concurrent validator thread calls searchGlobal() method to identify the source node in the BG. If any source node exists in the BG then it will do the same process as done by searchLocal(). After that validator thread calls the decInCount() to decreases the *inCnt* of all the conflicting nodes atomically which are present in the *eList* of corresponding source node. While decrementing the *inCnt* of each conflicting nodes in the BG, it again checks if any conflicting node became a source node then it adds that node into its local *cacheList* to optimize the search time of identifying the next source node. Due to lack of space, please refer accompanying technical report [1] to get the complete details with the algorithm of lock-free graph library methods.

B. Concurrent Miner

Smart contracts in blockchain are executed in two different context. First, by miner to propose a new block and after that by multiple validators to verify the block proposed by miner. In this subsection, we describe how miner executes the smart contracts concurrently. ① *Concurrent miner* gets the set of transactions from the *distributed shared memory* as shown in Figure 3. Each transaction associated with the functions (or atomic-units) of smart contracts. To run the smart contracts concurrently we have faced the challenge to identify the conflicting transactions at run-time because smart contract language are Turing-complete. Two transactions are in conflict if they are accessing common shared data-objects and at least one of them perform write operation on it. ② In *concurrent miner*, conflicts are identified at run-time with the help of efficient framework provided by optimistic software transactional memory system (STMs). STMs access the shared data-objects called as *t-object*s. Each shared *t-object* having initial state (or IS) which modified by the atomic-units and change IS to some other valid state. Eventually, it reaches to

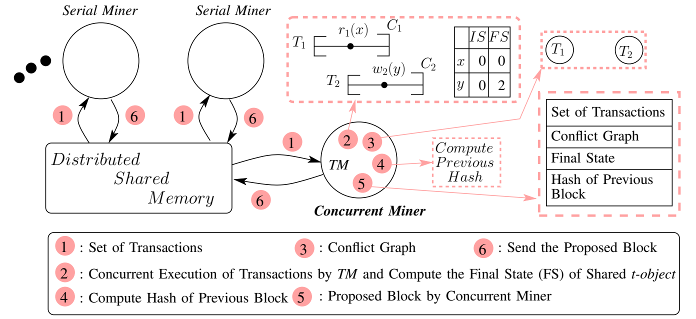


Fig. 3. Execution of Concurrent Miner

final state (or FS) at the end of block creation. As shown in Algorithm 2, first, each transaction T_i gets the unique timestamp i from STM.begin() at Line 6. Then transaction T_i executes the atomic-unit of smart contracts. *Atomic-unit* consists of multiple steps such as $read$ and $write$ on shared *t-object*s as x. Internally, these $read$ and $write$ steps are handled by the STM.read() and STM.write(), respectively. At Line 10, if current atomic-unit step (or curStep) is $read(x)$ then it calls the STM.read(x). Internally, STM.read() identify the shared *t-object* x from transactional memory (or TM) and validate it. If validation is successful then it gets the value as v at Line 11 and execute the next step of atomic-unit otherwise re-execute the atomic-unit if v is $abort$ at Line 12.

If curStep is $write(x)$ at Line 15 then it calls the STM.write(x). Internally, STM.write() stores the information corresponding to the shared *t-object* x into transaction local log (or *txlog*) in write-set (or $wset_i$) for transaction T_i. We use an optimistic approach in which effect of the transaction will reflect onto the TM after the successful STM.tryC(). If validation is successful for all the $wset_i$ of transaction T_i in STM.tryC() i.e. all the changes made by the T_i is consistent then it updates the TM otherwise re-execute the atomic-unit if v is $abort$ at Line 24. After successful validation of STM.tryC(), it also maintains the conflicting transaction of T_i into conflict list in TM.

③ Once the transaction commits, it stores the conflicts in the block graph (or BG). To maintain the BG it calls addVert() and addEdge() methods of the lock-free graph library. The internal details of addVert() and addEdge() methods are explained in SubSection IV-A. ④ Once the transactions successfully executed the atomic-units and completed with the construction of BG then *concurrent miner* compute the hash of the previous block. Eventually, ⑤ *concurrent miner* propose a block which consists of set of transactions, BG, final state of each shared *t-object*s, hash of the previous block of the blockchain and ⑥ send it to all other existing node in the *distributed shared memory* to validate it as shown in Figure 3.

C. Concurrent Validator

Concurrent validator validates the block proposed by the concurrent miner. It executes the set of transactions concurrently and deterministically with the help of block graph given by the *concurrent miner*. BG consists of dependency among the

Algorithm 2 Concurrent Miner(*auList[]*, STM): Concurrently m threads are executing atomic-units of smart contract from *auList[]*(or list of atomic-units) with the help of STM.

```
1:  procedure CONCURRENT MINER(auList[], STM)
2:      curAU ← curInd.get&Inc(auList[]);
3:      /*curAU is the current atomic-unit taken from the auList[] */
4:      /*Execute until all the atomic-units successfully completed*/
5:      while (curAU < size_of(auList[])) do
6:          T_i ← STM.begin();/*Create a new transaction T_i with timestamp i*/
7:          while (curAU.steps.hasNext()) do /*Assume that curAU is a list of steps*/
8:              curStep = currAU.steps.next(); /*Get the next step to execute*/
9:              switch (curStep) do
10:             case read(x):
11:                 v ← STM.read_i(x); /*Read t-object x from a shared memory*/
12:                 if (v == abort) then
13:                     goto Line 6;
14:                 end if
15:             case write(x, v):
16:                 /*Write t-object x into T_i local memory with value v*/
17:                 STM.write_i(x, v);
18:             case default:
19:                 /*Neither read from or write to a shared memory t-object*/
20:                 execute curStep;
21:         end while
22:         /*Try to commit the current transaction T_i and update the confList[i]*/
23:         v ← STM.tryC_i();
24:         if (v == abort) then
25:             goto Line 6;
26:         end if
27:         Create vNode with ⟨i, AU_id, 0, nil, nil⟩ as a vertex of Block Graph;
28:         BG(vNode, STM);
29:         curAU ← curInd.get&Inc(auList[]);
30:     end while
31: end procedure
```

conflicting transactions that restrict them to execute serially whereas non-conflicting transactions can run concurrently. In *concurrent validator* multiple threads are executing the atomic-units of smart contracts concurrently by *executeCode()* method at Line 38 and Line 45 with the help of searchLocal(), and searchGlobal() and decInCount() methods of lock-free graph library at Line 37, Line 44 and (Line 40, Line 47) respectively. The functionality of these lock-free graph library methods are explained in SubSection IV-A.

After the successful execution of all the atomic-units, *concurrent validator* compares its computed final state of each shared data-objects with the final states given by the *concurrent miner*. If the final state matches for all the shared data-objects then the block proposed by the *concurrent miner* is valid. Finally, the block is appended to the blockchain and respective *concurrent miner* is rewarded.

Theorem 2: All the dependencies between the conflicting nodes are captured in the BG.

V. EXPERIMENTAL EVALUATION

For the experiment, we consider a set of benchmarks generated for Ballot, Simple Auction, and Coin contracts from Solidity documentation [4]. Experiments are performed by varying the number of atomic-units, and threads. The analysis focuses on two main objectives: (1) Evaluate and analyzes the speedup achieved by concurrent miner over the serial miner. (2) Appraise the speedup achieved by concurrent validator over serial validator on various experiments.

Experimental system: The Experimental system is a large-scale 2-socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60 GHz

Algorithm 3 Concurrent Validator(*auList[]*, BG): Concurrently V threads are executing atomic-units of smart contract with the help of BG given by the miner.

```
32: procedure CONCURRENT VALIDATOR(auList[], BG)
33:     /*Execute until all the atomic-units successfully completed*/
34:     while (nCount < size_of(auList[])) do
35:         while (cacheList.hasNext()) do /*First search into thread local cacheList */
36:             cacheVer ← cacheList.next();
37:             cacheVertex ← searchLocal(cacheVer, AU_id);
38:             executeCode(AU_id); /*Execute the atomic-unit of cacheVertex*/
39:             while (cacheVertex) do
40:                 cacheVertex ← decInCount(cacheVertex);
41:             end while
42:             Remove the current node (or cacheVertex) from local cacheList;
43:         end while
44:         vexNode ← searchGlobal(BG, AU_id); /*Search into the BG*/
45:         executeCode(AU_id); /*Execute the atomic-unit of vexNode*/
46:         while (verNode) do
47:             verNode ← decInCount(verNode);
48:         end while
49:     end while
50: end procedure
```

with 14 cores per socket and two hyper-threads (HTs) per core, for a total of 56 threads. The machine has 32GB of RAM and runs Ubuntu 16.04.2 LTS.

Methodology: We have considered two types of workload, (W1) The number of atomic-units varies from 50 to 400, while threads and shared data-objects are fixed to 50 and 40 respectively. (W2) The number of threads varies from 10 to 60 while atomic-units are fixed to 400 and shared data-objects to 40. In all the experiments time taken by miners and validators is collected as an average of ten executions for the final result.

A. Benchmarks

In reality, miner forms a block which consists of a set of transactions from different contracts. So, we consider four benchmarks Ballot, Simple Auction, Coin including Mixed contract which is the combination of above three. In Ethereum blockchain, smart contracts are written in Solidity and runs on the Ethereum Virtual Machine (EVM). The issue with EVM is that it does not support multi-threading and hence give poor throughput. Therefore, to exploit the efficient utilization of multi-core resources and to improve the performance, we convert smart contract from Solidity language into C++ and execute them using multi-threading. The details of the benchmarks are as follows:

1) **Simple Auction:** It is an auction contract in which *bidders, highestBidder*, and *highestBid* are the shared data-objects. A single owner initiates the auction after that bidders can bid in the auction. The termination condition for auction is the bidding period (or end time) initialized at the beginning of the auction. During bidding period multiple bidders initiate their bids with biding amount using *bid()* method. At the end of the auction, a bidder with the highest amount will be successful, and respective bid amount is transferred to the beneficiary account. Conflict can occur if at least two bidders are going to request for bidPlusOne() simultaneously.
2) **Coin:** It is the simplest form of a cryptocurrency in which accounts are the shared data-objects. All accounts

are uniquely identified by Etherum addresses. Only the contract deployer known as minter will be able to generate the coins and initialize the accounts at the beginning. Anyone having an account can send coins to another account with the condition that they have sufficient coins in their account or can check their balance. In the initial state, minter initializes all the accounts with some coins. Conflict can occur if at least two senders are transferring the amount into the same receiver account simultaneously or when one send() and getbalance() have an account in common.

3) **Ballot:** It implements an electronic voting contract in which voters and proposals are the shared data-objects. All the voters and proposals are already registered and have unique Ethereum address. At first, all the voters are given rights by the chairperson (or contract deployer) to participate in the ballot. Voters either cast their vote to the proposal of their choice or delegate vote to another voter whom they can trust using delegate(). A voter is allowed to delegate or vote once throughout the ballot. Conflict can occur if at least two voters are going to delegate their vote to the same voter or cast a vote to the same proposal simultaneously. Once the ballot period is over, the winner of the ballot is decided based on the maximum vote count.
4) **Mixed:** In this benchmark, we have combined all the above benchmarks in equal proportions. Data conflicts occur when atomic-units of the same contract executed simultaneously, and operate on common shared data-objects.

In all the above contracts, conflicts can very much transpire when *miner* executes them concurrently. So, we use Optimistic STMs to ensure consistency and handle the conflicts.

B. Results

We have shown the speedup of concurrent execution by miner and validator over serial in Table I. The results from the serial execution of the miner and validator are served as the baseline.

Figure 4 and Figure 5 represent the speedup of concurrent miner and validator relative to the serial miner and validator for all the smart contracts on workload W1 and W2. It shows average speedup of 3.6x and 3.7x by the BTO and MVTO concurrent miner over serial miner respectively. Along with, BTO and MVTO validator outperforms average 40.8x and 47.1x than serial validator respectively [a]. The maximum speedup by concurrent miner on workload W1 is achieved at the smaller number of atomic-units. On workload W2 speedup of concurrent miner increases while increasing the number of threads up to fix number depending on system configuration.

The time taken by the concurrent validator is negligible as compared to serial validator because concurrent validator executes contracts concurrently and deterministically using BG given by concurrent miner. BG simplifies the parallelization task for the validator as validator need not to determine the conflicts, and directly executes non-conflicting transactions concurrently. It is clear from Figure 4 and Figure 5 that BTO and MVTO

[a] Code is available here: https://github.com/pdcrl/Blockchain

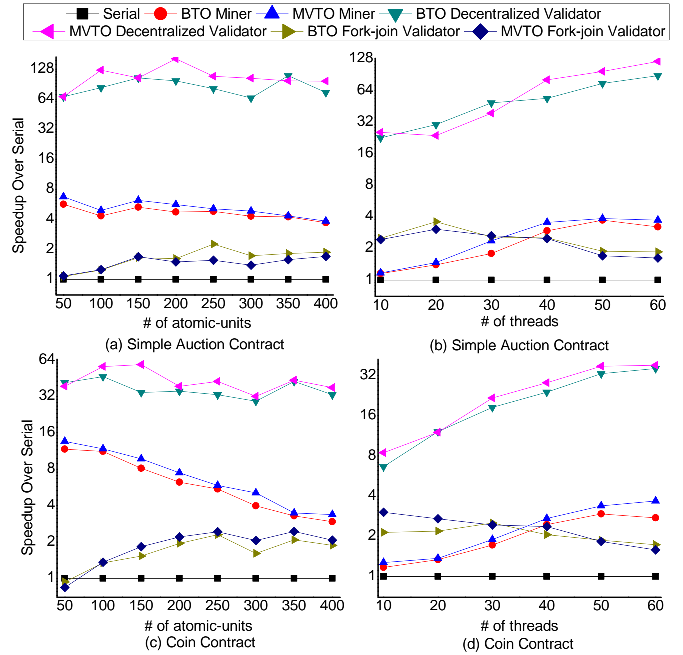


Fig. 4. Simple Auction and Coin Contracts

TABLE I
SPEEDUP ACHIEVED BY CONCURRENT MINER AND VALIDATOR

	Simple Auction		Coin		Ballot		Mixed	
	W1	W2	W1	W2	W1	W2	W1	W2
BTO Miner	4.6	2.4	6.6	2.1	3.8	3.1	4.8	1.6
MVTO Miner	5.2	2.7	7.5	2.4	2.3	1.5	5.7	1.8
BTO Decentralized Validator	85.7	53.1	36.9	21.7	126.7	152.1	90.7	68.6
MVTO Decentralized Validator	108.5	64.6	43.5	24.4	135.8	180.8	109.5	67.4
BTO Fork-join Validator	1.7	2.5	1.7	2.1	2.1	3.8	1.5	1.9
MVTO Fork-join Validator	1.5	2.3	1.9	2.3	1.8	3.8	1.5	2.7

Decentralized Validator is giving far better performance than BTO and MVTO Fork-join Validator. A possible reason can be master thread of BTO and MVTO Fork-join Validator becomes slow to assign the task to slave threads.

Figure 4 shows the speedup achieved by concurrent MVTO Miner is greater than BTO Miner for Simple Auction and Coin contract on workload W1 and W2 respectively. A plausible reason can be that MVTO gives good performance for read-intensive workloads [11]. Here, Simple Auction and Coin contracts are read-intensive [4]. Figure 4 (c) represents the speedup achieved by BTO and MVTO Fork-join Validators are even less than serial for 50 AUs due to the overhead of allocating the task by master thread.

Figure 5 (a) and (b) capture better speedup achieved by BTO Miner as compare to MVTO Miner for workload W1 and W2 because Ballot contract is write-intensive [4]. Figure 5 (c) and (d) represent the speedup achieved by concurrent miner and validator over serial miner and validator for the Mixed contract on workload W1 and W2 respectively. Due to equal proportions of all the above three contracts, the Mixed contract becomes read-intensive. So, the properties of the Mixed contract are same as Simple Auction and Coin contract with similar reasoning. Due to space constraints, we present essential results in the main paper and the remaining results on different workloads

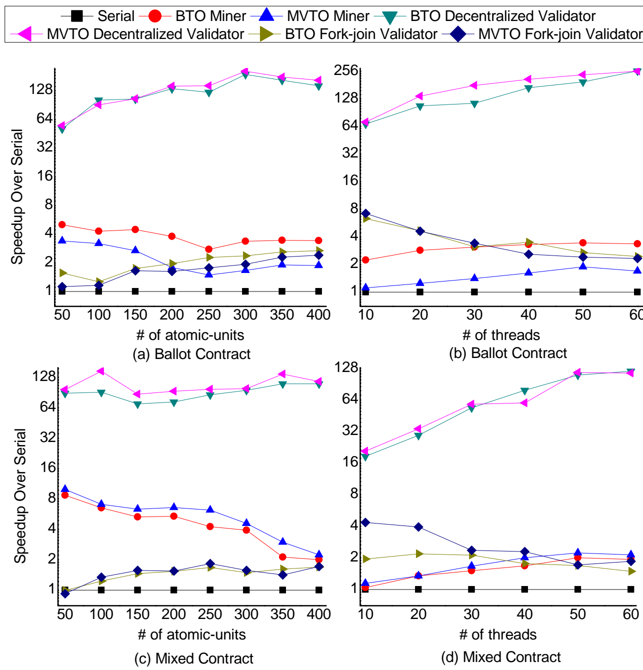


Fig. 5. Ballot and Mixed Contracts

are available in the accompanying technical report [1].

VI. Conclusion

To exploit the multi-core processors, we have proposed the concurrent execution of smart contract by miners and validators which improves the throughput. Initially, miner executes the smart contracts concurrently using optimistic STM protocol as BTO. To reduce the number of aborts and improves the efficiency further, the concurrent miner uses MVTO protocol which maintains multiple versions corresponding to each data-object. Concurrent miner proposes a block which consists of a set of transactions, BG, hash of the previous block and final state of each shared data-objects. Later, the validators re-execute the same smart contract transactions concurrently and deterministically with the help of BG given by miner which capture the conflicting relations among the transactions to verify final state. If the validation is successful then proposed block appended into the blockchain and miner gets incentive otherwise discard the proposed block. Overall, BTO and MVTO miner performs 3.6x and 3.7x speedups over serial miner respectively. Along with, BTO and MVTO validator outperform average 40.8x and 47.1x than serial validator respectively.

Acknowledgements. This project is in part supported by a research grant from Thynkblynk Technologies Pvt. Ltd, and IMPRINT India scheme. We are grateful to Dr. Bapi Chatterjee, Dr. Sandeep Hans, Mr. Ajay Singh for several useful discussions that we had on this topic. We would like to thank anonymous reviewers for their useful comments. We are also very grateful to Anila Kumari and G Monika the developers of IITH STM.

References

[1] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. *CoRR*, 2018. URL: https://arxiv.org/abs/1809.01326.

[2] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016.*

[3] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. PODC '17.

[4] Solidity documentation. URL: http://solidity.readthedocs.io/en/latest/index.html.

[5] Ethereum. URL: http://github.com/ethereum.

[6] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *PPoPP*, pages 175–184. ACM, 2008.

[7] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory*. Morgan and Claypool, 2010.

[8] Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[9] Maurice Herlihy and Nir Shavit. On the nature of progress. OPODIS 2011.

[10] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[11] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A TimeStamp Based Multi-version STM Algorithm. In *ICDCN*, pages 212–226, 2014.

[12] Petr Kuznetsov and Sathya Peri. Non-interference and local correctness in transactional memory. *Theor. Comput. Sci.*, 688:103–116, 2017.

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS '16*.

[14] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

[16] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[17] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *Financial Cryptography Workshops*, 2017.

[18] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, pages 204–213, 1995.

[19] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[20] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[21] An Zhang and Kunlong Zhang. Enabling concurrency on smart contracts using multiversion ordering. In Yi Cai, Yoshiharu Ishikawa, and Jianliang Xu, editors, *Web and Big Data*, pages 425–439, Cham, 2018.