Efficient Concurrent Execution of Smart Contracts in Blockchains Using Object-Based Transactional Memory

Parwat Singh Anjana[1], Hagit Attiya[2], Sweta Kumari[2], Sathya Peri[1], and Archit Somani[2](✉)

[1] Department of Computer Science and Engineering, IIT Hyderabad, Sangareddy, India
cs17mtech11014@iith.ac.in, sathya_p@cse.iith.ac.in

[2] Department of Computer Science, Technion, Haifa, Israel
{hagit,sweta,archit}@cs.technion.ac.il

Abstract. Several popular blockchains such as Ethereum execute *complex transactions* through user-defined scripts. A block of the chain typically consists of multiple *smart contract transactions (SCTs)*. To append a block into the blockchain, a miner executes these SCTs. On receiving this block, other nodes act as *validators*, who re-execute these SCTs as part of the consensus protocol to validate the block. In Ethereum and other blockchains that support cryptocurrencies, a miner gets an incentive every time such a valid block is successfully added to the blockchain. When executing SCTs sequentially, miners and validators fail to harness the power of multiprocessing offered by the prevalence of multi-core processors, thus degrading throughput. By leveraging multiple threads to execute SCTs, we can achieve better efficiency and higher throughput. Recently, *Read-Write Software Transactional Memory Systems (RWSTMs)* were used for concurrent execution of SCTs. It is known that *Object-based STMs (OSTMs)*, using higher-level objects (such as hash-tables or lists), achieve better throughput as compared to RWSTMs. Even greater concurrency can be obtained using *Multi-Version OSTMs (MVOSTMs)*, which maintain multiple versions for each shared data item as opposed to *Single-Version OSTMs (SVOSTMs)*.

This paper proposes an efficient framework to execute SCTs concurrently based on object semantics, using *optimistic* SVOSTMs and MVOSTMs. In our framework, a multi-threaded miner constructs a *Block Graph (BG)*, capturing the *object-conflicts* relations between SCTs, and stores it in the block. Later, validators re-execute the same SCTs concurrently and deterministically relying on this BG.

A malicious miner can modify the BG to harm the blockchain, e.g., to cause *double spending*. To identify malicious miners, we propose *Smart Multi-threaded Validator (SMV)*. Experimental analysis shows that proposed multi-threaded miner and validator achieve significant performance gains over state-of-the-art SCT execution framework.

Keywords: Blockchain · Smart contract · Concurrency · Object-based Software Transactional Memory · Multi-version · Opacity · Conflict-opacity

C. Georgiou and R. Majumdar (Eds.): NETYS 2020, LNCS 12129, pp. 77–93, 2021.
https://doi.org/10.1007/978-3-030-67087-0_6

1 Introduction

Blockchains like Bitcoin [15] and Ethereum [2] have become very popular. Due to their usefulness, they are now considered for automating and securely storing user records such as land sale documents, vehicle, and insurance records. *Clients*, external users of the system, send requests to nodes to execute on the blockchain, as *smart contracts transactions (SCTs)*. An SCT is similar to the methods of a class in an object-oriented langugage, which encode business logic relating to the contract [4,8]. Listing 1 shows a smart contract function, transfer() of coin contract from Solidity [4]. It transfers the amount from sender to receiver if the sender has a sufficient balance.

Listing 1: Transfer function

```
transfer(s_id, r_id, amt){
  if(amt > bal[s_id])
      throw;
  bal[s_id] -= amt;
  bal[r_id] += amt;
}
```

Blocks are added to the blockchain by *block-creator* nodes also known as *miners*. A miner m packs several SCTs received from (possibly different) clients, to form a block B. Then, m executes the SCTs of the block sequentially to obtain the final state of the blockchain, which it stores in the block. To maintain the chain structure, m adds the hash of the previous block to the new block B and proposes B to be added to the blockchain.

On receiving the block B, other nodes act as *validators* that execute a global consensus protocol to decide the order of B in the blockchain. As part of the consensus protocol, validators re-execute all the SCTs of B sequentially to obtain the final state of the blockchain, assuming that B will be added to the blockchain. If the computed final state matches the one stored in B by the miner m then B is accepted by the validators. In this case, the miner m gets an incentive for adding B to the blockchain (in Ethereum and other cryptocurrency-based blockchains). Otherwise, B is rejected, and m does not get any reward. This execution is known as *order-execute model* [5] adapted by Ethereum and several other blockchains such as Bitcoin [15], EOS [1].

Previous Work: Dickerson et al. [8] observed that both miner and validators can execute SCTs concurrently to exploit multi-core processors. They observed another interesting advantage of concurrent execution of SCTs in Ethereum, where only the miner receives an incentive for adding a valid block while all the validators execute the SCTs in the block. Given a choice, it is natural for a validator to pick a block that supports concurrent execution and hence obtain higher throughput.

Concurrent execution of SCTs poses a challenge. Consider a miner m that executes the SCTs in a block concurrently. Later, a validator v may re-execute same SCTs concurrently, in an order that may yield a different final state than given by m in B. In this case, v incorrectly rejects the valid block B proposed by m. We denote this as *False Block Rejection* (*FBR*), noting that FBR may negate the benefits of concurrent execution.

Dickerson et al. [8] proposed a multi-threaded miner algorithm that is based on a *pessimistic Software Transactional Memory (STM)* and uses locks for synchronization between threads executing SCTs. STM [14,18] is a convenient concurrent programming interface for a programmer to access the shared memory using multiple threads. To avoid FBR, the miner identifies the *dependencies* between SCTs in the block while executing them by multiple threads. Two SCTs are *dependent* if they are *conflicting*, i.e., both of them access the same data item and at least one of them is a write. These dependencies among SCTs are recorded in the block as a *Block Graph (BG)*. Two SCTs that have a path in the BG are *dependent* on each other and cannot be executed concurrently. Later, a validator v relies on the BG to identify dependencies between the SCTs, and concurrently execute SCTs only if there is no path between them in the BG. In the course of the execution by v, the size of BG dynamically decreases and the dependencies change. Dickerson et al. [8] use a *fork-join* approach to execute the SCTs, where a master thread allocates independent SCTs to different slave threads to execute.

Anjana et al. [6] used an *optimistic* Read-Write STM (RWSTM), which identifies the conflicts between SCTs using timestamps. Those are used by miner threads to build the BG. A validator processes a block using the BG in a completely decentralized manner using multiple threads, unlike the centralized fork-join approach of [8]. Each validator thread identifies an independent SCTand executes it concurrently with other threads. The decentralized approach yields significant performances gain over fork-join.

Saraph and Herlihy [17] used a *speculative bin* approach to execute SCTs of Ethereum in parallel. A miner uses lock to store SCTs into two bins, *concurrent bin* stores non-conflicting SCTs while the *sequential bin* stores the remaining SCTs. If an SCT T_i requests a lock held by an another SCT T_j then T_i is rolled back and placed in the sequential bin. Otherwise, T_i is placed in the concurrent bin. To save the cost of rollback and retries of SCTs, they have used *static conflict prediction* which identifies conflicting SCTs before executing them speculatively. The multi-threaded validator in this approach executes all the SCTs of the concurrent bin concurrently and then executes the SCTs of the sequential bin sequentially. We call this the *Static Bin* approach.

Zhang and Zhang [20] proposed a pessimistic approach to execute SCTs concurrently in which the miner can use any concurrency control protocol while the validator uses *multi-version timestamp order*.

Exploiting Object-Based Semantics: Prior STM-based solutions of [6,20], rely on *read-write conflicts (rwconflicts)* for synchronization. In contrast, *object-based STMs (OSTMs)* track higher-level, more advanced conflicts between operations like insert, delete, lookup on a hash-table, enqueue/dequeue on queues, push/pop on the stack [11,12,16]. It has been shown that OSTMs provide greater concurrency than RWSTMs (see Fig. 1 in [7]). This is particularly important since Solidity [4], the language used for writing SCTs for Ethereum, extensively uses hash-tables. This indicates that a hash-table based OSTM is a natural candidate for concurrent execution of these SCTs.[1]

The pessimistic lock-based solution of [8] uses abstract locks on hash-table keys, exploiting the object semantics. In this paper, we want to exploit the object semantics of hash-tables using optimistic STMs to improve the performance obtained.

To capture the dependencies between the SCTs in a block, miner threads construct the BG concurrently and append it to the block. The dependencies between the transactions are given by the *object-conflicts (oconflicts)* (as opposed to rwconflicts) which ensure that the execution is correct, i.e., satisfies *conflict-opacity* [16]. It has been shown [11,12,16] that there are fewer oconflicts than rwconflicts. Since there are fewer oconflicts, the BG has fewer edges which in turn, allows validators to execute more SCTs concurrently. This also reduces the size of the BG leading to a smaller communication cost.

[1] For clarity, we denote smart contract transactions as SCTs and an STM transaction as a transaction in the paper.

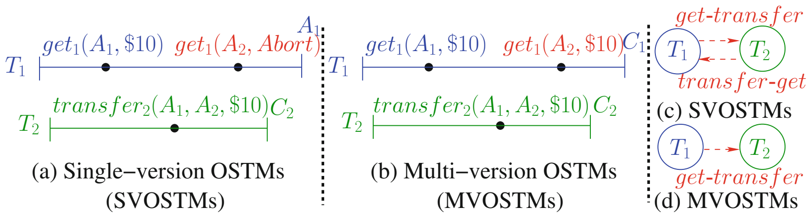

Fig. 1. (a) Transaction T_1 gets the balance of two accounts A_1 and A_2 (both initially \$10), while transaction T_2 transfers \$10 from A_1 to A_2 and T_1 aborts. Since, its conflict graph has a cycle (see (c)); (b) When T_1 and T_2 are executed by MVOSTM, T_1 can read the old versions of A_1 and A_2. This can be serialized, as shown in (d).

Multi-version object-based STMs (MVOSTMs) [13] maintain multiple versions for each shared data item (object) and provide greater concurrency relative to traditional *single-version OSTMs (SVOSTMs)*. Figure 1 illustrates the benefits of concurrent execution of SCTs using MVOSTM over SVOSTM. A BG based on MVOSTM will have fewer edges than an SVOSTM-based BG, and will further reduce the size of the BG. These advantages motivated us to use MVOSTMs for concurrent execution of SCTs by miners.

Concurrent executions of SCTs may cause inconsistent behaviors such as *infinite loops*, *divide by zero*, *crash failures*. Some of these behaviors, such as crash failures and infinite loops can be mitigated when SCTs are executed in a controlled environment, for example, the *Ethereum Virtual Machine (EVM)* [2]. However, not all environments can prevent all anomalies. The inconsistent executions can be prevented by ensuring that the executions produced by the STM system satisfy *opacity* [9] or one of its variants such as *co-opacity* [16]. Our MVOSTM satisfies opacity, while our SVOSTM satisfies co-opacity.

Handling a Malicious Miner: A drawback of the approaches mentioned above is that a malicious miner can make the final state of the blockchain be inconsistent. In the BG approach, the miner can send an incorrect BG, missing some edges. In the bin-based approach [17], the miner can place the conflicting transactions in the concurrent bin. This can result in inconsistent states in the blockchain due to *double spending*, e.g., when two concurrent transactions incorrectly transfer the same amount of money simultaneously from a source account to two different destination accounts. If a malicious miner does not add an edge between these two transactions in the BG [6] or puts them in the concurrent bin [17], then both SCTs can execute concurrently by validators. If a majority of validators accept the block containing these two transactions, then the state of the blockchain becomes inconsistent. We denote this problem as *edge missing BG* (*EMB*) for the BG approach [6] and *faulty bin* (*FBin*) for the bin-based approach [17]. In Sect. 4, we show the effect of malicious miners through experiments on the blockchain system.

To handle EMB and FBin errors, the validator must reject a block when edges are missing in the BG or when conflicting SCTs are in the concurrent bin, since their execution can lead to an inconsistent state. To detect this situation, validator threads monitor transactions performing conflicting access to the same data items while executing concurrently. In Sect. 3, we propose a *Smart Multi-threaded Validator (SMV)* which uses $counters$ to detect this condition and rejects the corresponding blocks.

Dickerson et al. [8] suggest a lock-based solution to handle EMB errors. The miner generates and stores the lock profile required to execute the SCTs of a block along with the BG. The validator then records a trace of the locks each of its thread would have acquired, had it been executing speculatively independent of the BG. The validator would then compare the lock profiles it generated with the one provided by the miner present in the block. If the profiles are different then the block is rejected. This check is in addition to the check of the final state generated and the state in the block. This solution is effective in handling EMB errors caused by malicious miners. However, it is lock-based and cannot be used for preventing EMB issue in optimistic approaches such as [6]. The advantage of our SMV solution is that it works well with both optimistic and lock-based approaches.

Our Contributions: This paper develops an efficient framework to execute SCTs concurrently by a miner using an optimistic hash-table (both single and multi-version) OSTM. We use two methodologies to re-execute the SCTs concurrently by validators: the *fork-join approach* [8] and a *decentralized approach* [6]. To handle EMB and FBin errors, we propose a decentralized *smart multi-threaded validator*. To summarize:

– We introduce an efficient object-based framework for the concurrent execution of SCTs by miners (Sect. 3.2). We propose a way to execute SCTs efficiently using optimistic SVOSTM by the miner while ensuring *co-opacity* [16], a way to execute SCTs by the miner using optimistic MVOSTM [13] while satisfying *opacity* [9]
– We propose the concurrent execution of SCTs by validators using the BG provided by the miner to avoid FBR errors (Sect. 3.3), using either the fork-join or the decentralized approach.
– We propose a Smart Multi-threaded Validator to handle EMB and FBin errors caused by malicious miners (Sect. 3.4).
– Extensive simulations (Sect. 4) show that concurrent execution of SCTs by SVOSTM and MVOSTM miner provide an average speedup of 3.41$\times$ and 3.91$\times$ over serial miner, respectively. SVOSTM and MVOSTM based decentralized validator provide on average of 46.35$\times$ and 48.45$\times$ over serial validator, respectively.

2 System Model

As in [10,14], in each miner/validator there are n threads, $p_1, \ldots, p_n$ in a system that access shared data items (or objects/keys) in a completely asynchronous fashion. We assume that none of the threads/processes will crash or fail unexpectedly.

Events: A thread invokes the transactions and the transaction calls object-level methods that internally invoke read/write atomic events on the shared data items to communicate with other threads. Method invocations (or inv) and responses (or rsp) are also considered as events.

History: It is a sequence of invocations and responses of different transactional methods. We consider *sequential history* in which invocation on each transactional method follows the immediate matching response. We consider *well-formed* histories in which a new transaction does not begin until the invocation of previous transaction has not been committed or aborted.

Object-Based Software Transactional Memory (OSTM): OSTM exports higher-level methods: (1) STM_begin(): begins a transaction with unique id. (2) STM_lookup(k) (or $l(k)$): does a lookup on data item k from shared memory. (3) STM_insert(k, v) (or $i(k, v)$): inserts the value of data item k as v in its local log. (4) STM_delete(k) (or $d(k)$): deletes the data item k. (5) STM_tryC(): validates the transaction. After successful validation, the actual effects of *STM_insert()* and *STM_delete()* will be visible in the shared memory and transaction returns commit ($\mathcal{C}$). Otherwise, it will return abort ($\mathcal{A}$). We represent *STM_lookup()*, and *STM_delete()* as *return-value (rv)* methods because both methods return the value from hash-table. We represent *STM_insert()*, and *STM_delete()* as *update* (*upd*) methods as on successful *STM_tryC()* both methods update the shared memory. Methods *rv()* and *STM_tryC*() may return $\mathcal{A}$. For a transaction T_i, we denote all the objects accessed by its $rv_i()$ and $upd_i()$ methods as $rvSet_i$ and $updSet_i$, respectively.

Listing 2: Transfer function using STM

```
7   transfer(s_id, r_id, amt){
8     t_id = STM_begin();
9     s_bal = STM_lookup(s_id);
10    if(amt => s_bal) {
11      abort(t_id);
12      throw;
13    }
14    STM_delete(s_id, amt);
15    STM_insert(r_id, amt);
16    if(STM_tryC(t_id)!= SUCCESS)
17      goto Line 8;//Trans aborted
18  }
```

Listing 2 shows the concurrent execution of transfer() (from Listing 1 in the Sect. 1) using STM. On the invocation of transfer(), STM assigns the unique id using *STM_begin()* to each SCT (Line 8). Then, it reads the balance of the sender using *STM_lookup()* (Line 9) and validates it (Line 10). If the sender does not have a sufficient balance, then it *aborts* the SCT and throws an exception. Otherwise, it withdraws the amount from the sender account using *STM_delete()* (Line 14) and deposits the amount in the receiver account using *STM_insert()* (Line 15). With an optimistic STM, the effect of the *STM_delete()* and *STM_lookup()* will take place after successful validation of the SCT in *STM_tryC()* (Line 16). If validation is successful, then the SCT commits, and the amount is transferred from the sender to the receiver account. Otherwise, the SCT is aborted and re-execute from Line 8.

Valid and Legal History: If the successful $rv_j(k, v)$ (i.e., $v \neq \mathcal{A}$) method of a transaction T_j *returns* the value from any of previously committed transaction T_i that has performed $upd()$ on key k with value v then such $rv_j(k, v)$ method is *valid*. If all the *rv()* methods of history H are valid then H is valid history [16].

If the successful $rv_j(k, v)$ (i.e., $v \neq \mathcal{A}$) method of a transaction T_j *returns* the value from previous closest committed transaction T_i that $k \in updSet_i$ (T_i can also be T_0) and updates the k with value v then such $rv_j(k, v)$ method is *legal*. If all the *rv()* methods of history H are legal then H is legal history [16]. A legal history is also valid.

Two histories H and H$'$ are *equivalent* if they have the same set of events. H and H$'$ are *multi-version view equivalent* [19, Chap. 5] if they are valid and equivalent. H and H$'$ are *view equivalent* [19, Chap. 3] if they are legal and equivalent. Additional definitions appear in [7].

3 Proposed Mechanism

This section describes the construction, data structures, and methods of concurrent BG, concurrent execution of SCTs by multi-threaded miner using optimistic object-based STMs, multi-threaded validator, and detection of a malicious miner.

3.1 The Block Graph

The multi-threaded miner executes the SCTs concurrently and stores their dependencies in a BG. Each committed transaction corresponding to an SCTis a vertex in the BG while edges capture the dependencies, based on the STM protocol. Multi-threaded miner uses SVOSTM or MVOSTM to execute the SCTs concurrently, using timestamps. The challenge here is to construct the BG concurrently without missing any dependencies. We modified SVOSTM and MVOSTM to capture *oconflicts* and *multi-version oconflicts* (*mvoconflicts*) in the BG.

SVOSTM-based miner maintains three types of edges based on oconflicts between the transactions. An edge $T_i \rightarrow T_j$ between two transaction is defined when: **(1)** $rv_i(k, v)$ - $STM\_tryC_j()$ $edge$: If $rv_i(k, v)$ on key k by T_i completed before $STM\_tryC_j()$ on k by a committed transaction T_j in history H such that T_i returns a value $v \neq \mathcal{A}$. Formally, $rv_i(k, v) <_H STM\_tryC_j()$, $k \in updSet(T_j)$ and $v \neq \mathcal{A}$; **(2)** $STM\_tryC_i()$ - $rv_j(k, v)$ $edge$: If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $rv_j(k, v)$ on key k by T_j in history H such that T_j returns a value $v \neq \mathcal{A}$. Formally, $STM\_tryC_i() <_H rv_j(k, v)$, $k \in updSet(T_i)$ and $v \neq \mathcal{A}$; **(3)** $STM\_tryC_i()$ - $STM\_tryC_j()$ $edge$: If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $STM\_tryC_j()$ on k by a committed transaction T_j in history H. Formally, $STM\_tryC_i() <_H STM\_tryC_j()$ and ($updSet(T_i)$ $\cap$ $updSet(T_j) \neq \emptyset$).

MVOSTM-based miner maintains two types of edges based on *mvoconflicts* [13]. **(1)** *return value from (rvf) edge:* If $STM\_tryC_i()$ on k by a committed transaction T_i completed before $rv_j(k, v)$ on key k by T_j in history H such that T_j returns a value $v \neq \mathcal{A}$ then there exist an *rvf edge* from T_i to T_j, i.e., $T_i \rightarrow T_j$; **(2)** *multi-version (mv) edge:* consider a triplet, $STM\_tryC_i(), rv_m(k, v)$, $STM\_tryC_j()$ in which ($updSet(T_i) \cap updSet(T_j) \cap rvSet(T_m) \neq \emptyset$), (two committed transactions T_i and T_j update the key k with value v and u respectively) and ($u, v \neq \mathcal{A}$); then there are two types of *mv edge*: (a) if $STM\_tryC_i() <_H STM\_tryC_j()$ then there exist a *mv edge* from T_m to T_j. (b) if $STM\_tryC_j() <_H STM\_tryC_i()$ then there exist a *mv edge* from T_j to T_i.

Data Structure for the Block Graph: To maintain a block graph $BG(V, E)$, the set of vertices (or SCTs) V is stored as a vertex list and the set of edges (conflicts between SCTs) E is stored as an adjacency list. Two lock-free methods build the BG (see details in [7]): *addVertex()* adds a vertex and *addEdge()* adds an edge in BG. To execute the SCTs, validator threads use three methods: *globalSearch*() identifies an independent vertex with indegree 0 to execute it concurrently, *remExNode*() decrements the indegree of conflicting vertices and keeps it into thread local log if its indegree becomes 0, and *localSearch*() identifies the vertex with indegree 0 in thread local log to execute it concurrently.

Algorithm 1. Multi-threaded Miner(sctList[], STM): n threads concurrently execute the SCTs from sctList with STMs.

```
19: procedure Multi-threaded Miner (sctList[], STM)
20:     $curInd = gIndex.get\&Inc()$; // Atomically read the index and increment it.
21:     while ($curInd < sctList.length$) do // Execute until all SCTs have not been executed
22:         $curTrn = sctList[curInd]$; // Get the current SCTto execute
23:         $T_i$ = STM_begin(); // Begins a new transaction. Here $i$ is unique id
24:         for all ($curStep \in curTrn.scFun$) do // scFun is a list of steps
25:             switch(curStep)
26:                case lookup($k$):
27:                    $v \leftarrow$ STM_lookup($k$); // Lookup data item $k$ from a shared memory
28:                    if($v == \mathcal{A}$) then goto Line 23;end if break;
29:                case insert($k$, $v$): // Insert data item k into $T_i$ local memory with value $v$
30:                    STM_insert($k$, $v$); break;
31:                case delete($k$):
32:                    $v \leftarrow$ STM_delete($k$); // Actual deletion of data item k happens in STM_tryC()
33:                    if($v == \mathcal{A}$) then goto Line 23; end if break;
34:                default: Execute the step normally // Any step apart from lookup, insert, delete
35:             endswitch
36:         end for
37:         $v \leftarrow$ STM_tryC(); // Try to commit the transaction $T_i$
38:         if($v == \mathcal{A}$) then goto Line 23; end if
39:         addVertex(i); // Create vertex node for $T_i$ with scFun
40:         BG(i, STMs); // Add the conflicts of $T_i$ to block graph
41:         $curInd = gIndex.get\&Inc()$; // Atomically read the index and increment it.
42:     end while
43:     build-block(); // Here the miner builds the block.
44: end procedure
```

3.2 Multi-threaded Miner

A miner m receives requests to execute SCTs from different clients. It forms a block with several SCTs (the precise number of SCTs depend on the blockchain), and executes these SCTs while executing the non-conflicting SCTs concurrently to obtain the final state of the blockchain. Identifying the non-conflicting SCTs statically is not straightforward because smart contracts are written in a turing-complete language [8] (e.g., Solidity [4] for Ethereum). We use optimistic STM to execute the SCTs concurrently as in [6] but adapted to object-based STMs on hash-tables to identify conflicts.

Algorithm 1 shows how SCTs are executed by an n-threaded miner. The input is an array of SCTs, $sctList$ and a object-based STM, (SVOSTM or MVOSTM), both supporting the BG methods described above. The multi-threaded miner uses a global index into the sctList $gIndex$ which is accessed by all the threads. A thread Th_x first reads the current value of $gIndex$ into a local value $curInd$ and increments $gIndex$ atomically (Line 20).

Having obtained the current index in $curInd$, Th_x gets the corresponding SCT, $curTrn$ from $sctList[]$ (Line 22), and begins a STM transaction corresponding to $curTrn$ (Line 23). For every hash-table insert, delete and lookup encountered while executing the scFun of $curTrn$, Th_x invokes the corresponding STM methods: *STM_lookup()*, *STM_insert()*, *STM_delete()*, either on an SVOSTM or on an MVOSTM. Otherwise, it simply executes the step. If any of these steps fail, Th_x begins a new STM transaction (Line 23) and re-executes these steps.

Upon successful completion of transaction T_i, Th_x creates a vertex node for T_i in the block graph (Line 39). Then, Th_x obtains the transactions (SCTs) with which T_i is conflicting from the OSTM, and adds the corresponding edges to the BG (Line 40). Th_x then gets the index of the next SCTto execute (Line 41).

An important step here is how the underlying OSTMs (either SVOSTM or MVOSTM) maintain the conflicts among the transactions which is used by Th_x (see [7]). Both SVOSTM and the MVOSTM use timestamps to identify the conflicts.

Once all the SCTs of sctList have been executed successfully and the BG is constructed concurrently, it is stored in the proposed block. The miner then stores the final state of the blockchain (which is the state of all shared data items), resulting from the execution of SCTs of sctList in the block. The miner then computes the operations related to the blockchain. For Ethereum, this would constitute the hash of the previous block. Then the multi-threaded miner proposes a block which consists of all the SCTs, BG, final state of all the shared data items and hash of the previous block (Line 43). The block is then broadcast to all the other nodes in the blockchain.

We prove the next properties (see [7]):

Theorem 1. *The BG captures all the dependencies between the conflicting nodes.*

Theorem 2. *A history H_m generated by the multi-threaded miner with SVOSTM satisfies co-opacity.*

Theorem 3. *A history H_m generated by multi-threaded miner with MVOSTM satisfies opacity.*

3.3 Multi-threaded Validator

The validator re-executes the SCTs deterministically relying on the BG provided by the miner in the block. BG consists of dependency among the conflicting SCTs and restrict validator threads to execute them serially to avoid the FBR errors while non-conflicting SCTs execute concurrently to obtain greater throughput. The validator uses *globalSearch*(), *localSearch*(), and *remExNode*() methods of the BG library as described in Sect. 3.1.

After successful execution of the SCTs, validator threads compute the final state of the blockchain which is the state of all shared data items. If it matches the final state provided by the miner then the validator accepts the block. If a majority of the validators accept the block, then it is added to the blockchain. Detailed description and proofs of the next theorems appear in [7].

Theorem 4. *A history H_m generated by the multi-threaded miner with SVOSTM and a history H_v generated by a multi-threaded validator are view equivalent.*

Theorem 5. *A history H_m generated by the multi-threaded miner with MVOSTM and a history H_v generated by a multi-threaded validator are multi-version view equivalent.*

3.4 Detection of Malicious Miners by Smart Multi-threaded Validator (SMV)

We propose a technique to handle edge missing BG (EMB) and Faulty Bin (FBin) caused by the malicious miner as explained in Sect. 1. A malicious miner mm can remove some edges from the BG and set the final state in the block accordingly. A multi-threaded validator executes the SCTs concurrently relying on the BG provided by the mm and results the same final state. Hence, incorrectly accepts the block. Similarly, if a majority of the validators accept the block then the state of the blockchain becomes inconsistent. For example, due to double spending.

A similar inconsistency can be caused by a mm in bin-based approach: mm can maliciously add conflicting SCTs to the concurrent bin resulting in FBin error. This may cause a multi-threaded validator v to access shared data items concurrently leading to synchronization errors. To prevent this, an SMV checks to see if two concurrent threads end up accessing the same shared data item concurrently. If this situation is detected, then the miner is malicious.

Algorithm 2. SMV(scFun): Execute scFun with atomic global lookup/update counter.

```
45: while (scFun.steps.hasNext()) do //scFun is a list of steps
46:     curStep = scFun.steps.next(); //Get the next step to execute
47:     switch (curStep) do
48:     case lookup($k$):
49:         if ($k.gUC$ == $k.lUC_i$) then //Check for update counter ($uc$) value
50:             Atomically increment the global lookup counter, $k.gLC$;
51:             Increment $k.lLC_i$ by 1. //Maintain $k.lLC_i$ in transaction local log
52:             Lookup $k$ from a shared memory;
53:         else return ⟨Miner is malicious⟩;
54:         end if
55:     case insert($k$, $v$):
56:         if (($k.gLC$ == $k.lLC_i$) && ($k.gUC$ == $k.lUC_i$)) then //Check lookup/update counter value
57:             Atomically increment the global update counter, $k.gUC$;
58:             Increment $k.lUC_i$ by 1. //Maintain $k.lUC_i$ in transaction local log
59:             Insert $k$ in shared memory with value $v$;
60:         else return ⟨Miner is malicious⟩;
61:         end if
62:     case delete($k$):
63:         if (($k.gLC$ == $k.lLC_i$) && ($k.gUC$ == $k.lUC_i$)) then //Check lookup/update counter value
64:             Atomically increment the global update counter, $k.gUC$;
65:             Increment $k.lUC_i$ by 1. //Maintain $k.lUC_i$ in transaction local log
66:             Delete $k$ in shared memory.
67:         else return ⟨Miner is malicious⟩;
68:         end if
69:     case default:
70:         curStep is not lookup, insert and delete;
71:         execute curStep;
72: end while
73: Atomically decrement the $k.gLC$ and $k.gUC$ corresponding to each shared data-item key $k$.
```

To identify such situations, an SMV uses $counters$, inspired by the *basic timestamp ordering (BTO)* protocol in databases [19, Chap. 4]. It tracks each global data item that can be accessed across multiple transactions by different threads. Specifically, the SMV maintains two global counters for each key of hash-table (shared data item) k (a) $k.gUC$ - global update counter (b) $k.gLC$ - global lookup counter. These, respectively, track the number of **updates** and **lookups** that are concurrently performed by different threads on k. Both counters are initially 0.

When an SMV thread Th_x is executing an SCT T_i it maintains two local variables corresponding to each global data item k which is accessible only by Th_x (c) $k.lUC_i$ - local update counter (d) $k.lLC_i$ - local lookup counter. These respectively keep track of number of updates and lookups performed by Th_x on k while executing T_i. These counters are initialized to 0 before the start of T_i.

Having described the counters, we will explain the SMV Algorithm 2 at a high level. Suppose the next step to be performed by Th_x is:

1. $lookup(k)$: Thread Th_x will check for equality of the global and local update counters, i.e., $(k.gUC == k.lUC_i)$ (Line 49). If they are not same then SMV will report the miner as malicious (Line 53). Otherwise, (i) Th_x will atomically increment $k.gLC$ (Line 50). (ii) Th_x will increment $k.lLC_i$ (Line 51). (iii) Perform the lookup on the key k from shared memory (Line 52).
2. $update(k, val)$: Here, Th_x wants to update (insert/delete) k with value val. So, Th_x will check for the equality of both global, local update and lookup counters, i.e., $(k.gUC == k.lUC_i)$ and $(k.gLC == k.lLC_i)$ (Line 56 or Line 63). If they are not same then SMV will report the miner as malicious (Line 60 or Line 67). Otherwise, (i) Th_x will atomically increment $k.gUC$ (Line 57 or Line 64). (ii) Th_x will increment $k.lUC_i$ (Line 58 or Line 65). (iii) Update key k with value val in the shared memory (Line 59 or Line 66).

Once T_i terminates, Th_x will atomically decrements $k.gUC, k.gLC$ by the value of $k.lUC_i, k.lLC_i$, respectively (Line 73). Then Th_x will reset $k.lUC_i, k.lLC_i$ to 0.

The reason for performing these steps and the correctness of the algorithm is as follows: if Th_x is performing a lookup on k then no other thread should be performing an update on k. Here, $k.gUC$ represents the number of updates to k currently executed by all the threads while $k.lUC_i$ represents the number of updates to k on behalf of T_i by Th_x. Thus the value of gUC should be same as lUC. Otherwise, some other thread is also concurrently performing the updates to k. Similarly, if Th_x is performing an update on k, then no other thread should be performing an update or lookup on k. This can be verified by checking if lLC, lUC are respectively same as gLC, gUC.

Theorem 6. *Smart Multi-threaded Validator rejects malicious blocks with BG that allow concurrent execution of dependent SCTs.*

The same SMV technique can be applied to identify the *faulty bin* error as explained in Sect. 1. See proof of Theorem 6 in [7].

4 Experimental Evaluation

This section demonstrates the performance gains by proposed multi-threaded miner and validator against state-of-the-art miners and validators. To evaluate our approach, we considered Ethereum smart contracts. The virtual environment of Ethereum, EVM, does not support multi-threading [2,8]. So, we converted the smart contracts of Ethereum as described in Solidity documentation [4] into C++ multi-threaded contracts similar to [6]. Then we integrated them into object-based STM framework (SVOSTM and MVOSTM) for concurrent execution of SCTs by the miner.

We chose a diverse set of smart contracts described in Solidity [4] as benchmarks to analyze the performance of our proposed approach as was done in [6,8]. The selected benchmark contracts are (1) *Coin*: a financial contract, (2) *Ballot*: an electronic voting contract, (3) *Simple Auction*: an auction contract, and (4) a *Mix* contract: combination of three contracts mentioned above in equal proportion in which block consists of multiple SCTs belonging to different smart contracts.

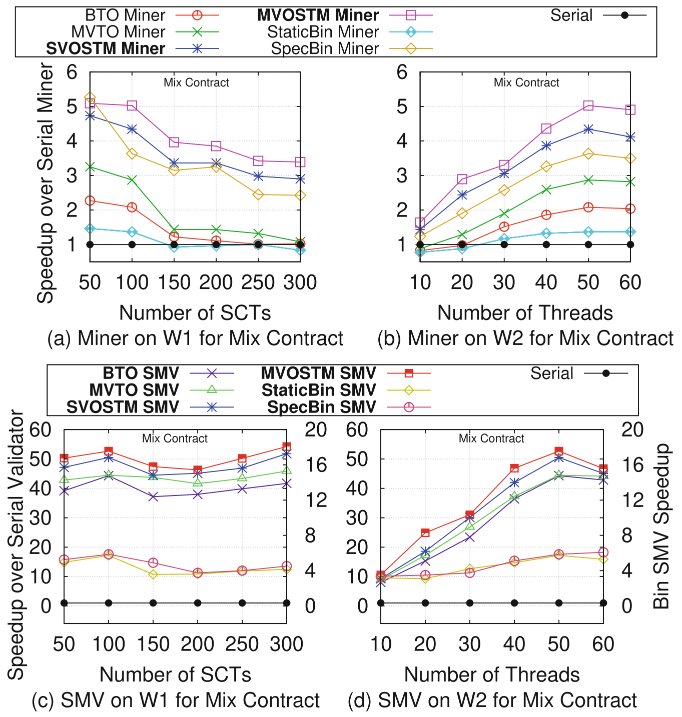

Fig. 2. Multi-threaded and SMVs Speedup over Serial Miner and Validator for Mix Contract on W1 and W2

We compared the proposed SVOSTM and MVOSTM miner with state-of-the-art multi-threaded: BTO [6], multi-version timestamp order (MVTO) [6], Speculative Bin (or SpecBin) [17], Static Bin (or StaticBin) [17], and Serial miner.[2] We could not compare our work with Dickerson et al. [8] as their source code is not available in public domain. We converted the code of StaticBin and SpecBin [17] from Java to C++ for comparing with our algorithms.

Concurrent execution of SCTs by the validator does not use any STM protocol; however it uses the BG provided by the multi-threaded miner, which does use STM. To identify malicious miners and prevent any malicious block from being added to the blockchain, we proposed Smart Multi-threaded Validator (SMV) for SVOSTM, MVOSTM as SVOSTM SMV, MVOSTM SMV. Additionally, we proposed SMV for state-of-the-art validators as BTO SMV, MVTO SMV, SpecBin SMV, and StaticBin SMV and analysed the performance.

Experimental Setup: The experimental system consists of two sockets, each comprised of 14 cores 2.60 GHz Intel (R) Xeon (R) CPU E5-2690, and each core supports 2 hardware threads. Thus the system supports a total of 56 hardware threads. The machine runs Ubuntu 16.04.2 LTS operating system and has 32GB RAM.

[2] Code is available in: https://github.com/PDCRL/ObjSC.

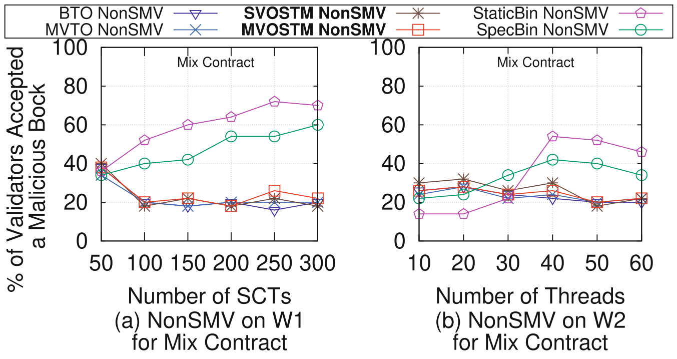

Fig. 3. % of average multi-threaded validator (NonSMV) accepted a malicious block for Mix Contract on W1 and W2

To analyze the performance, we evaluated the speedup achieved by each contract on two workloads. In the first workload (W1), the number of SCTs varied from 50 to 300 while the number of threads fixed is at 50. The maximum number of SCTs in a block of Ethereum is approximately 250 [3,8], but is growing over time. In the second workload (W2), the number of threads varied from 10 to 60, while the number of SCTs is fixed at 100. The average number of SCTs in a block of Ethereum is around 100 [3]. The hash-table size and shared data items are fixed to 30 and 500 respectively for both workloads. For accuracy, results are averaged over 26 runs in which the first run is discarded and considered as a warm-up run. The results of serial execution is treated as the baseline for evaluating the speedup. This section describes the detailed analysis for the Mix contract and analysis of Coin, Ballot and Simple Auction benchmark contracts are in [7].

Experimental Results: Fig. 2(a) and (b) show the speedup of MVOSTM, SVOSTM, MVTO, BTO, SpecBin, and StaticBin miner over serial miner for Mix contract on workloads W1 and W2, respectively.[1] The average speedup achieved by MVOSTM, SVOSTM, MVTO, BTO, SpecBin, and StaticBin miner over serial miner is 3.91×, 3.41×, 1.98×, 1.5×, 3.02×, and 1.12×, respectively.

As shown in Fig. 2(a), increasing the number of SCTs leads to high contention (because shared data items are fixed to 500). So the speedup of multi-threaded miner reduces. MVOSTM and SVOSTM miners outperform SpecBin miner because MVOSTM and SVOSTM miners use optimistic object-based STMs to execute SCTs concurrently and construct the BG whereas SpecBin uses locks to execute SCTs concurrently and constructs two bins using the pessimistic approach. SpecBin miner does not release the locks until the construction of the concurrent bin, which gives less concurrency. However, for the smaller numbers of SCTs in a block, SpecBin is slightly better than MVOSTM and SVOSTM miners, which can be observed in the Fig. 2(a) at 50 SCTs. MVOSTM and SVOSTM miners outperform MVTO and BTO miners because both of them are consider rwconflicts. It can also be observed that MVOSTM miner

[1] In the figures, legend items in bold.

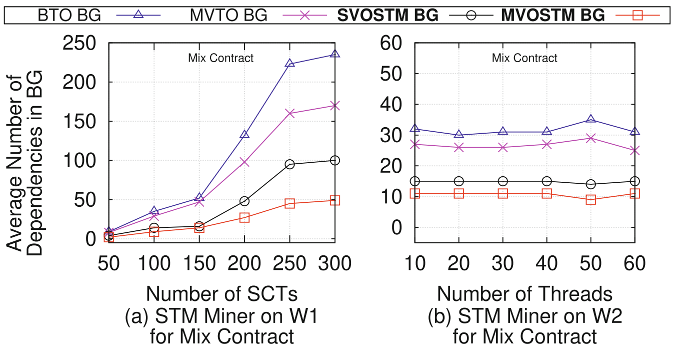

Fig. 4. Average Number of Dependencies in BG for Mix Contract on W1 and W2

outperforms all other STM miners as it has fewer conflicts, which gets reflected (see Fig. 4) as the least number of dependencies in the BG as compared to other STM miners. For the multi-version (MVOSTM and MVTO) miners, we did not limit the number of versions because the number of SCTs in a block is finite. The speedup by StaticBin miner is worse than serial miner for more than 100 SCTs because it takes time for *static conflict prediction* before executing SCTs.

Figure 2(b) shows that speedup achieved by multi-threaded miner increases while increasing the number of threads, limited by the number of hardware threads available on the underlying experimental setup. Since, our system has 56 logical threads, the speedup decreases beyond 56 threads. MVOSTM miner outperforms all other miners with similar reasoning, as explained for Fig. 2(a). Another observation is that when the number of threads is less, the serial miner dominates BTO and MVTO miner due to the overhead of the STM system.

The average number of dependencies in BG by all the STM miners presented in Fig. 4. It shows that BG constructed by the MVOSTM has the least number of edges for all the contracts on both workloads. However, there is no BG for bin-based approaches (both SpecBin and StaticBin). So, from the block size perspective, bin-based approaches are efficient. But the speedup of the validator obtained by the bin-based approaches is significantly lesser than STM validators.

Figure 2(c) and (d) show the speedup of Smart Multi-threaded Validators (SMVs) over serial validator on the workloads W1 and W2, respectively. The average speedup achieved by MVOSTM, SVOSTM, MVTO, BTO, SpecBin, and StaticBin decentralized SMVs are 48.45×, 46.35×, 43.89×, 41.44×, 5.39×, and 4.81× over serial validator, respectively.

It can be observed that decentralized MVOSTM SMV is best among all other STM validators due to fewer dependencies in the BG. Though the block size is less in bin-based approaches as compared to STM based approaches due to the absence of BG, however, STM validators outperform bin-based validators because STM validators precisely determines the concurrent SCTs based on BG. In contrast, bin-based validator gives less concurrency using a lock-based pessimistic approach.

The speedup of SMV is significantly higher than multi-threaded miner because the miner has to execute the SCTs concurrently either using STMs (including the retries of aborted transactions) and constructs the BG or prepare two bins (concurrent and sequential bin using locks in SpecBin and static analysis in StaticBin). On the other hand, the validator executes the SCTs concurrently and deterministically relying on BG (without any retries) or bins provided by miner.

A malicious miner may cause either EMB or FBin errors in a block. Figure 3 illustrates the percentage of validators without SMV logic embedded, i.e., NonSMVs accepting a malicious block on workloads W1 and W2, respectively. Here, we considered 50 validators and ran the experiments for the Mix contract. The Fig. 3 shows that less than 50% of validators (except bin-based NonSMV) accept a malicious block. However, SpecBin and StaticBin NonSMVs show more than 50% acceptance of malicious blocks. Though, it is to be noted that the acceptance of even a single malicious block result in the blockchain going into inconsistent state.

To solve this problem, we developed a Smart Multi-threaded Validator (SMV), which identifies the malicious miner (described in Sect. 3.4). We prove that the SMV detects malicious block with the help of $counter$ and rejects it. In fact all the validators shown in Fig. 2 (c) and (d) are SMV based. Another advantage of SMV is that once it detects a malicious miner during the concurrent execution of SCTs, it can immediately reject the block and need not execute the remaining SCTs in the block thus saving time.

To show the degree of parallelism, we consider *diameter* of BG which shows the longest path of the BG implies that a longest sequence of transactions to be executed sequentially. To observe the diameter of BG, we consider another workload W3 in which the number of shared data items varied from 100 to 600 while the number of threads, SCTs, and hash-table size is fixed to 50, 100, and 30, respectively. In Figure 5, Y1 axis shows the speedup achieved by SMV over serial and Y2 axis demonstrates the diameter of the BG in considered STMs. It shows that highest speedup achieved when diameter of the BG is least.

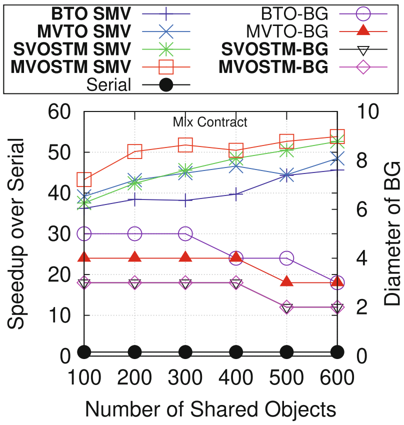

Fig. 5. Speedup of SMV over serial and Diameter of BG

We presents additional experiments that cover the average number of dependencies in the BG, additional space required to store the BG into the block, compared the time taken by the SMV and NonSMV, and speedup of fork-join validator for all the workloads in [7].

5 Conclusion and Future Directions

This paper presents an efficient framework for concurrent execution of smart contracts by miners and validators based on object semantics. In blockchains that follow order-execute model [5] such as Ethereum [2] and Bitcoin [15], SCTis executed in two different contexts: first by the multi-threaded miner to propose a block and later by the multi-threaded validator to verify the proposed block by the miner as part of the consensus. To avoid FBR errors, the miner on concurrent execution of SCTs capture the dependencies among them in the form of a BG as in [6,8]. The validator then re-executes the SCTs concurrently while respecting the dependencies recorded in the BG to avoid FBR errors.

The miner executes the SCTs concurrently using STMs that exploit the object semantics: SVOSTM and MVOSTM. The dependencies among the SCTs are collected during this execution and used by the miner threads to construct the BG concurrently. Due to the use of object semantics, the number of edges in the BG is smaller, which benefits both miners and validators by enabling them to execute SCTs quickly in a concurrent setting.

We also considered a malicious miner, which may proposes an incorrect BG that does not have all the edges, resulting in EMB error. To handle malicious miners we have proposed a SMV that can identify these errors and reject the corresponding blocks.

The proposed SVOSTM and MVOSTM miner achieve on average speedup of 3.41$\times$ and 3.91$\times$ over a serial miner, respectively. Proposed SVOSTM and MVOSTM decentralized validator outperform with an average speedup of 46.35$\times$ and 48.45$\times$ over serial validator, respectively, on Ethereum smart contracts.

There are several directions for future work. A malicious miner can intentionally append a BG in a block with additional edges for the purpose of delaying other miners. Preventing such a malicious miner from doing this would be an immediate future work. A natural question is whether the size of BG can become a significant overhead. Currently, the average number of SCTs in a block is $\approx$100 in Ethereum. So, storing the BG inside the block does not consume much space. The BG constructed by MVOSTMs has fewer dependencies as compared with state-of-the-art SCT execution as shown in Fig. 4. However, the number of SCTs in a block can increase over time and as a result storing the BG will consume more space. Hence, constructing storage optimal BG is an interesting challenge. Alternatively, it might be possible to concurrently execute SCTs correctly without incurring any extra storage overhead, and without compromising speedup. This opens up the question what the optimal storage required for achieving the best possible speedup.

Another interesting research direction is optimizing power consumption, since, multi-threading on the multi-core system consumes more power. Additional power is consumed by the multiple miner and validator threads to propose and validate the blocks concurrently. Hence, we would like to explore trade-off between harnessing the number of cores and power consumption.

Finally, since EVM [2] does not support multi-threading, it is not possible to test the proposed approach on Ethereum. So, another research direction is to design multi-threaded EVM. We plan to test our proposed approach on other blockchains such as Bitcoin [15], EOS [1] which follow the order-execute model and support multi-threading.

Acknowledgment. We are thankful to the anonymous reviewers for helpful suggestions. This research was supported by ISF grant 380/18, IMPRINT Project number 6918F funded by MHRD, and MEITY project number 4(20)/2019-ITEA.

References

1. EOS. https://eos.io/. Accessed 08 Mar 2020
2. Ethereum. http://github.com/ethereum. Acessed 08 Mar 2020
3. Ethereum Stats. https://etherscan.io. Accessed 08 Mar 2020
4. Solidity Documentation. https://solidity.readthedocs.io/. Accessed 08 Mar 2020
5. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. EuroSys (2018)
6. Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An efficient framework for optimistic concurrent execution of smart contracts. In: PDP, pp. 83–92 (2019)
7. Anjana, P.S., Attiya, H., Kumari, S., Peri, S., Somani, A.: Achieving greater concurrency in execution of smart contracts using object semantics. CoRR ArXiv:1904.00358 (2019). http://arxiv.org/abs/1904.00358
8. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding Concurrency to Smart Contracts, pp. 303–312. In: PODC. ACM, NY, USA, New York (2017)
9. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP (2008)
10. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory (2010)
11. Hassan, A., Palmieri, R., Ravindran, B.: Optimistic Transactional Boosting, vol. 49, pp. 387–388. ACM, New York, NY, USA 2014)
12. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: PPoPP (2008)
13. Juyal, C., Kulkarni, S., Kumari, S., Peri, S., Somani, A.: An innovative approach to achieve compositionality efficiently using multi-version object based transactional systems. In: Izumi, T., Kuznetsov, P. (eds.) SSS'2018, pp. 284–300. Springer, Cham (2018)
14. Kuznetsov, P., Peri, S.: Non-interference and local correctness in transactional memory. Theor. Comput. Sci. **688**, 103–116 (2017)
15. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2009)
16. Peri, S., Singh, A., Somani, A.: Efficient means of achieving composability using transactional memory. In: NETYS (2018)
17. Saraph, V., Herlihy, M.: An empirical study of speculative concurrency in ethereum smart contracts. In: Tokenomics (2019)
18. Shavit, N., Touitou, D.: Software transactional memory. In: PODC (1995)
19. Weikum, G., Vossen, G.: Transactional Info Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery (2002)
20. Zhang, A., Zhang, K.: Enabling concurrency on smart contracts using multiversion ordering. In: Web and Big Data (2018)