OptSmart: A Space Efficient Optimistic Concurrent Execution of Smart Contracts*

Parwat Singh Anjana†, Sweta Kumari‡, Sathya Peri†, Sachin Rathor†, and Archit Somani‡

†*Department of CSE, Indian Institute of Technology Hyderabad, Telangana, India*
‡*Department of Computer Science, Technion, Israel*

Abstract

Popular blockchains such as Ethereum and several others execute complex transactions in blocks through user-defined scripts known as *smart contracts*. Serial execution of smart contract transactions/atomic-units (AUs) fails to harness the multiprocessing power offered by the prevalence of multi-core processors. By adding concurrency to the execution of AUs, we can achieve better efficiency and higher throughput.

In this paper, we develop a concurrent miner that proposes a block by executing the AUs concurrently using *optimistic Software Transactional Memory systems (STMs)*. It captures the independent AUs in a *concurrent bin* and dependent AUs in the *block graph (BG)* efficiently. Later, we propose a concurrent validator that re-executes the same AUs concurrently and deterministically using a concurrent bin followed by BG given by the miner to verify the block. We rigorously prove the correctness of concurrent execution of AUs and show significant performance gain than state-of-the-art.

Keywords: Blockchain, Smart Contracts, Software Transactional Memory System, Multi-version, Concurrency Control, Opacity

*A preliminary version of this paper appeared in 27^{th} Euromicro International Conference On Parallel, Distributed, and Network-Based Processing (PDP[1]) 2019, Pavia, Italy. A poster version of this work received **Best Poster Award** in ICDCN 2019 [2].

**This manuscript covers the exhaustive related work, detailed proposed mechanism with algorithms, optimizations on the size of the block graph, rigorous correctness proof, and additional experimental evaluations with state-of-the-art.

***Author sequence follows lexical order of last names.

Email address: cs17mtech11014@iith.ac.in, sweta@cs.technion.ac.il, sathya_p@cse.iith.ac.in, cs18mtech01002@iith.ac.in, archit@cs.technion.ac.il (Parwat Singh Anjana†, Sweta Kumari‡, Sathya Peri†, Sachin Rathor†, and Archit Somani‡)

1. 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 that store the information in cryptographically linked blocks. Cryptocurrencies such as Bitcoin [3] and Ethereum [4] were the first to popularize the blockchain technology. Blockchains are now considered for automating and securely storing user records such as healthcare, financial services, real estate, etc. Blockchain network consists of multiple peers (or nodes) where 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.

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 [5]. 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. Ethereum uses Solidity [6] while Hyperledger supports language such as Java, Golang, Node.js, etc.

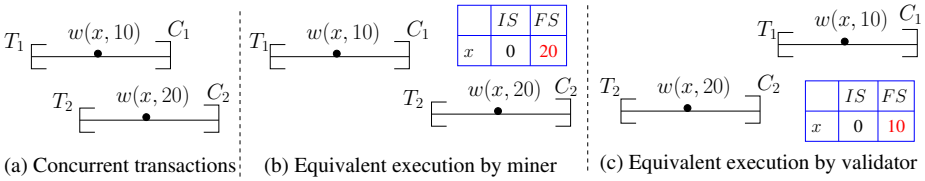
Motivation for Concurrent Execution of Smart Contracts: Dickerson et al. [5] observed that smart contract transactions are executed in two different contexts in Ethereum blockchain. First, executed by miners while forming a block– a miner selects a sequence of client requests (transactions), executes 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, and the previous block hash in the block. After creating the block, the miner proposes it to be added to the blockchain through the consensus protocol. The other peers in the system, referred to as *validators* in this context, validate the block

proposed by the miner. They re-execute the smart contract transactions in the block *serially* to verify the block's final states. If the 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. It has been observed that the validation runs several times more than the miner code [5].

This design of smart contract execution is not efficient as it does not allow any concurrency. In today's world of multi-core systems, the serial execution does not utilize all the cores, resulting in lower throughput. This limitation is not specific only to Ethereum blockchain but also applies to other popular blockchains as well. Higher throughput means more transaction execution per unit time, which clearly will be desired by both miners and validators.

However, the concurrent execution of smart contract transactions is not straightforward. Because various transactions could consist of conflicting access to the shared data objects. Two contract transactions are said to be in *conflict* if both of them access a shared data object, and at least one performs a write operation. Arbitrary execution of these smart contract transactions by the miners might result in the data-races leading to the inconsistent final state of the blockchain. Unfortunately, it is impossible to statically identify conflicting contract transactions since contracts are developed in Turing-complete languages. The common solution for correct execution of concurrent transactions is to ensure that the execution is *serializable* [7]. A usual correctness-criterion in databases, serializability ensure that the concurrent execution is equivalent to some serial execution of the same transactions. Thus miners must ensure that their execution is serializable [5] 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 is serializable. Later a validator re-executes the same transactions concurrently. However, during the concurrent execution, the validator may execute two conflicting transactions in an order different from the miner. Thus the serialization order of the miner is different from the validator. These 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



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

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. (b) represents the miner's concurrent execution 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 (c) shows the concurrent execution by a validator with an equivalent serial schedule as T_2, T_1, and the final state as 10 from IS 0, which is different from the final state proposed by the miner. Such a situation leads to the rejection of the valid block by the validator, which is undesirable.

although it is valid as depicted in Figure 1.

Dickerson et al. [5] identified these issues and proposed a solution for concurrent execution by both miners and validators. The miner concurrently executes block transactions using abstract locks and inverse logs to generate a serializable execution. Then, to enable correct concurrent execution by the validators, the miners provide a *happen-before* graph in the block. The happen-before graph is a direct acyclic graph over all the transactions 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 happen-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 happen-before graph in the block provides a greater enhancement to validators to consider such blocks. It helps them execute quickly through parallelization instead of a block that does not have any parallelization tools. This fascinates the miners to provide such tools in the block for concurrent execution by the validators.

Proposed Solution Approach - Optimistic Concurrent Execution and Lock-Free Graph: Dickerson et al. [5] 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 propose a *novel* and *efficient* framework for concurrent miner using *optimistic* Software Transactional Memory Systems (STMs). STMs are suitable for the concurrent executions of transactions without worrying about consistency issues.

The requirement of the miner, is to concurrently execute the smart contract transactions correctly and output a graph capturing dependencies among the transactions of the block such as happen-before graph. We denote this graph as *block graph* (or BG). The miner uses an optimistic STM system to execute the smart contract transactions concurrently in the proposed solution. 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 the 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 execute atomically (or serialized). To differentiate between smart contract transaction from STM transaction, we denote smart contract transaction as *atomic-unit* (*AU*) and STM transaction as *transaction* in the rest of the document. Thus the miner uses the STM system to invoke a transaction for each AU. In case the transaction gets aborted, then the STM repeatedly invokes new transactions for the same AU until a transaction invocation eventually commits.

A popular correctness guarantee provided by STM systems is *opacity* [8] 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, 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-objects*).

Due to simplicity, we have chosen two timestamp based STMs in our design: (1) *Basic Timestamp Ordering* or *BTO* STM [9, Chap 4], maintains only one version for

each *t-object*. (2) *Multi-Version Timestamp Ordering* or *MVTO* STM [10], 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 the equivalent serial history is ordered based on the transactions' timestamps. Thus using the timestamps, the miner can generate the BG of the AUs. We call it as *STM approach*. Dickerson et al. [5], developed the BG in a serial manner. Saraph and Herlihy [11] proposed a simple *bin-based two-phase speculative* approach to execute AUs concurrently in the Ethereum blockchain without storing the BG in the block. We analyzed that the bin-based approach reduces the size of the block but fails to exploits the concurrency. We name this approach as *Speculative Bin* (Spec Bin) approach. So, in our proposed approach, we combined spec bin-based approach [11] with the STM approach [1] for the optimal storage of BG in a block and exploit the concurrency. Concurrent miner generates an efficient BG in concurrent and lock-free [12] manner.

The concurrent miner applies the STM approach to generate two bins while executing AUs concurrently, a concurrent bin and a sequential bin. AUs which can be executed concurrently (without any conflicts) are stored in the concurrent bin. While the AUs having conflicts are stored in a sequential bin in the BG form to record the conflicts. This combined technique reduces the size of the BG than [1] while storing the graph of only sequential bin AUs instead of all AUs.

We propose a concurrent validator that creates multiple threads. Each of these threads parses the concurrent bin followed by efficient BG provided by the concurrent miner and re-execute the AUs for validation. The BG consists of only dependent AUs. Each validator thread claims a node that does not have any dependency, i.e., a node without any incoming edges by marking it. After that, it executes the corresponding AUs deterministically. Since the threads execute only those nodes with no incoming edges, the concurrently executing AUs 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* as the multiple threads are working on BG concurrently in the absence of a master thread.

The approach adopted by Dickerson et al. [5], works on *fork-join* in which a master

thread allocates different tasks to slave threads. The master thread identifies AUs that do not have any incoming dependencies in the BG and allocates them to different slave threads. In this paper, we compare the performance of both these approaches with the serial validator.

The significant contributions of the paper are as follows:

- Introduce a novel way to execute the AUs by concurrent miner using *optimistic* STMs (Section 4). We implement the concurrent miner using BTO and MVTO STM, but it is generic to any STM protocol.
- We propose a *lock-free* and concurrent graph library to generate the *efficient* BG which contains only dependent atomic-units and optimize the size of the block than [1] (see Section 4).
- We propose concurrent validator that re-executes the AUs deterministically and efficiently with the help of *concurrent bin* followed by *efficient* BG given by concurrent miner (see Section 4).
- To make our proposed approach storage optimal and efficient, we have optimized the BG size (see Section 4).
- We rigorously prove that the concurrent miner and validator satisfies correctness criterion as *opacity* (see Section 5).
- We achieve $4.49\times$ and $5.21\times$ average speedups for optimized concurrent miner using BTO and MVTO STM protocol, respectively. Optimized concurrent BTO and MVTO decentralized validator outperform average $7.68\times$ and $8.60\times$ than serial validator, respectively (Section 6).

Section 2 presents the related work on concurrent execution of smart contract transactions. While, Section 3 includes the notions related to STMs and execution model used in the paper. The conclusion with several future directions is presented in Section 7.

2. Related Work

This section presents the related work on concurrent execution on blockchains in line with the proposed approach.

Table 1: Related Work Summary

	Miner Approach	Locks	Require Block Graph	Validator Approach	Blockchain Type
Dickerson et al. [5]	Pessimistic ScalaSTM	Yes	Yes	Fork-join	Permissionless
Zhang and Zhang [17]	-	-	Read, Write Set	MVTO Approach	Permissionless
Anjana et al. [1]	Optimistic RWSTM	No	Yes	Decentralized	Permissionless
Amiri et al. [18]	Static Analysis	-	Yes	-	Permissioned
Saraph and Herlihy [11]	Bin-based Approach	Yes	No	Bin-based	Permissionless
Anjana et al. [19]	Optimistic ObjectSTM	No	Yes	Decentralized	Permissionless
Proposed Approach	**Bin+Optimistic RWSTM**	**No**	**No (if no dependencies) / Yes**	**Decentralized**	**Permissionless**

The interpretation of *Blockchain* was introduced by Satoshi Nakamoto in 2009 as Bitcoin [3] to perform electronic transactions without third party interference. Nick Szabo [13] introduced smart contracts in 1997, adopted by Ethereum blockchain in 2015 to expand blockchain functionalities beyond financial transactions (cryptocurrencies). A smart contract is an interface to reduce the computational transaction cost and provides secure relationships on distributed networks. There exist several papers [14, 15, 16] in the literature that works on the safety and security concern of smart contracts, which is out of the scope of this paper. We mainly focus on the concurrent execution of AUs. A concise summary of closely related works is given in Table 1.

Dickerson et al. [5] introduced concurrent executions of AUs in the blockchain. They observed that miners and validators could execute AUs simultaneously to exploit concurrency offered by ubiquitous multi-core processors. The approach of this work is given in Section 1.

Zhang and Zhang [17] proposed a concurrent miner using a pessimistic concurrency control protocol, which delays the read until the corresponding writes to commit and ensures a conflict-serializable schedule. The proposed concurrent validator uses MVTO protocol to execute transactions concurrently using the write sets provided by the concurrent miner in the block.

Anjana et al. [1] proposed optimistic *Read-Write STM* (RWSTM) using BTO and MVTO based protocols. The timestamp-based protocols are used to identify the conflicts between AUs. The miner executes the AUs using RWSTM and constructs the BG dynamically at the runtime using the timestamps. Later, a concurrent *Decentralized Validator* (Dec-Validator) executes the AUs in the block in a decentralized manner. The Decentralized Validator is efficient than the Fork-Join Validator since there is no

master validator thread to allocate the AUs to the slave validator threads to execute. Instead, all the validator threads identify the source vertex (a vertex with indegree 0) in the BG independently and claim the source node to execute the corresponding AU.

Amiri et al. [18] proposed *ParBlockchain*– an approach for concurrent execution of transactions in the block for permissioned blockchain. They developed an *OXII paradigm*[1] to support distributed applications. The OXII paradigm orders the block transactions based on the agreement between the orderer nodes using static analysis or speculative execution to obtain the read-set and write-set of each transaction, then generates the BG and constructs the block. The executors from respective applications (similar to the executors in fabric channels) execute the transactions concurrently and then validate them by re-executing the transaction. So, the nodes of the ParBlockchain execute the transactions in two phases using the OXII paradigm. A block with BG based on the transaction conflicts is generated in the first phase, known as the *ordering phase*. The second phase, known as the *execution phase*, executes the block transactions concurrently using the BG appended with block.

Saraph and Herlihy [11] proposed a simple *bin-based two-phase speculative* approach to execute AUs concurrently in the Ethereum blockchain. They empirically validated the possible benefit of their approach by evaluating it on historical transactions from the Ethereum. In the first phase, the miner uses locks and executes AUs in a block concurrently by rolling back those AUs that lead to the conflict(s). All the aborted AUs are then kept into a sequential bin and executed in the second phase sequentially. The miner gives concurrent and sequential bin hints in the block to the validator to execute the same schedule as executed by the miner. The validator executes the concurrent bin AUs concurrently while executes the sequential bin AUs sequentially. Instead of BG, giving hints about bins takes less space. However, it does not harness the maximum concurrency available within the block.

Later, Anjana et al. [19] proposed an approach that uses optimistic single-version and multi-version *Object-based STMs (OSTMs)* for the concurrent execution of AUs by

[1] A paradigm in which transactions are first ordered for concurrent execution then executed by both miners and validators [18].

the miner. The OSTMs operate at a higher (object) level rather than page (read-write) level and constructs the BG. However, the BG is still quite significantly large in the existing approaches and needs higher bandwidth to broadcast such a large block for validation.

In contrast, we propose an efficient framework for concurrent execution of the AUs using optimistic STMs. We combine the benefits of both Spec Bin-based and STM-based approaches to optimize the storage aspects (efficient storage optimal BG), which further improves the performance. Due to its optimistic nature, 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 [8]. The proposed approach gives better speedup over state-of-the-art and serial execution of AUs.

3. System Model

In this section, we will present the notions related to STMs and the execution model used in the proposed approach.

Following [20, 21], we assume a system of n processes/threads, $p_1, \ldots, p_n$ that access a collection of *transactional objects* or *t-object*s via atomic *transactions*. Each transaction has a unique identifier. Within a transaction, processes can perform *transactional operations or methods*:

- `STM.begin()` – begins a transaction.
- `STM.write`(x, v) (or $w(x, v)$)– updates a *t-object* x with value v in its local memory.
- `STM.read`(x, v) (or $r(x, v)$)– tries to read x and returns value as v.
- `STM.tryC`()– tries to commit the transaction and returns $commit$ (or $\mathcal{C}$) if succeeds.
- `STM.tryA`()– 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-objects* 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 consider *sequential* histories, i.e., 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 $\langle evts(H), <_H \rangle$.

Further, we consider *well-formed* histories, 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* [9, 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* [9, 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* [9, 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) [9, 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 [9, 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 [7]. To

circumvent this issue, researchers in databases have identified an efficient sub-class 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) [9, Chap. 3], if there exist a serial history S such that S is conflict equivalent to H.

Serializability and Opacity: Serializability [7] 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 [8]. Opacity, on the other hand, considers the correctness of *aborted* transactions as well.

A history H is said to be *opaque* [8, 20] 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. Similar to view-serializability, verifying the membership of opacity is NP-Complete [7]. To address this issue, researchers have proposed another popular correctness-criterion *co-opacity* whose membership is polynomial time verifiable.

Co-opacity: A history H is said to be *co-opaque* [21] 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., $\prec_H^{RT} \subset \prec_S^{RT}$.

(3) S preserves conflicts (i.e. $\prec_H^{Conf} \subseteq \prec_S^{Conf}$).

Linearizability: A history H is linearizable [22] if

(1) The invocation and response events can be reordered to get a valid sequential history.

(2) The generated sequential history satisfies the object's 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 [12] if the program threads are

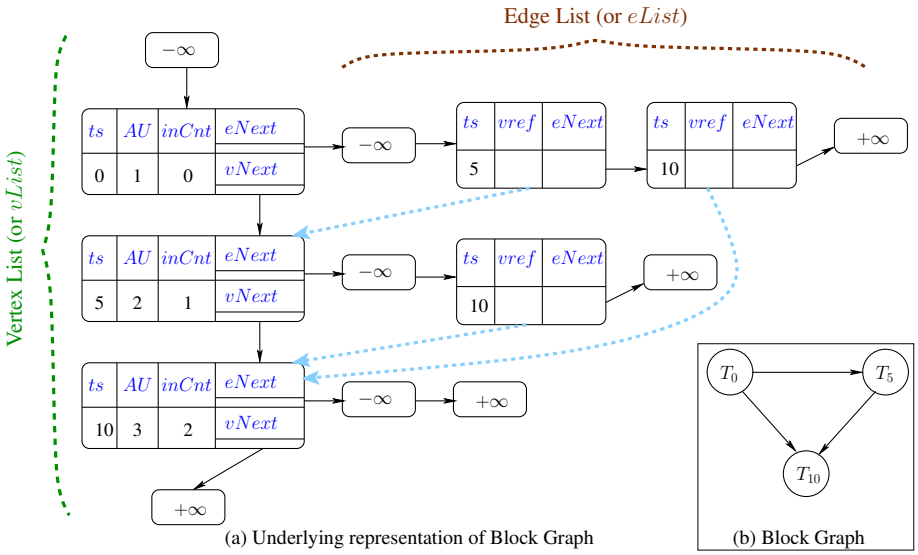


Figure 2: Pictorial representation of Block Graph

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.

4. Proposed Mechanism

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

4.1. Lock-free Concurrent Block Graph

Data Structure of Lock-free Concurrent Block Graph: We use the *adjacency list* to maintain the block graph BG(V, E), as shown in Figure 2 (a). Where V is a set of vertices (or *vNodes*) 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 $\langle ts = i, AU_{id} = id, \textit{inCnt} = 0, \textit{vNext} = nil, \textit{eNext} = nil \rangle$. Where i is a unique timestamp (or ts) of transactions T_i. AU_{id} is the id of a atomic-unit executed by transaction T_i. To maintain the indegree count of each *vNode*, we initialize *inCnt* as 0. *vNext* and *eNext* initialize as nil.

While E is a set of edges which maintains all 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 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 having a conflict with T_i and $ts(T_i)$ is less than $ts(T_j)$. We add conflicting edges from lower timestamp to higher timestamp transactions to maintain the acyclicity in the BG 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. To make it search efficient, *vertex node reference (or vref)* keeps the reference of its own vertex which is present in the *vList* and *eNext* initializes as nil.

The block graph (BG) generated by the concurrent miner 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: The 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, it first calls the `addVert()` method. The `addVert()` method identifies the correct location of that node (or *vNode*) in the *vList* at Line 16. If *vNode* is not part of *vList*, it creates the node and adds it into *vList* at Line 19 in a lock-free manner using atomic compare and swap (CAS) operation. Otherwise, *vNode* is already present in *vList* at Line 24.

Algorithm 1 BG(*vNode*, STM): It generates a BG for all the atomic-unit nodes.

```
1:  procedure BG(vNode, STM)
2:      /*Get the confList of transaction T_i from STM*/
3:      clist ← STM.getConfList (vNode.ts_i);
4:      /*T_i conflicts with T_j and T_j existes in conflict list
        of T_i*/
5:      for all (ts_j ∈ clist) do
6:          addVert (ts_j);
7:          addVert (vNode.ts_i);
8:          if (ts_j < vNode.ts_i) then
9:              addEdge (ts_j, vNode.ts_i);
10:         else
11:             addEdge (vNode.ts_i, ts_j);
12:         end if
13:     end for
14: end procedure
```

Algorithm 2 `addVert`(ts_i): It adds the vertex in the BG for T_i.

15: **procedure** `addVert`(ts_i)
16: Identify ⟨*vPred*, *vCurr*⟩ of *vNode* of ts_i in *vList*;
17: **if** (*vCurr*.$ts_i \neq$ *vNode*.ts_i) **then**
18: Create new Graph Node (*vNode*) of ts_i in *vList*;
19: **if** (*vPred*.*vNext*.CAS(*vCurr*, *vNode*)) **then**
20: return⟨*Vertex added*⟩;
21: **end if**
22: goto Line 16; /*Start with the *vPred* to identify the new ⟨*vPred*, *vCurr*⟩*/
23: **else**
24: return⟨*Vertex already present*⟩;
25: **end if**
26: **end procedure**

Algorithm 3 `addEdge`(*fromNode, toNode)*: It adds an edge from *fromNode* to *toNode*.

27: **procedure** `addEdge`(*fromNode, toNode)*
28: Identify the ⟨*ePred*, *eCurr*⟩ of *toNode* in *eList* of the *fromNode* vertex in BG;
29: **if** (*eCurr*.$ts_i \neq$ toNode.ts_i) **then**
30: Create new Graph Node (or *eNode*) in *eList*;
31: **if** (*ePred*.*eNext*.CAS(*eCurr*, *eNode*)) **then**
32: Increment the *inCnt* atomically of *eNode*.*vref* in *vList*;
33: return⟨*Edge added*⟩;
34: **end if**
35: goto Line 28; /*Start with the *ePred* to identify the new ⟨*ePred*, *eCurr*⟩*/
36: **else**
37: return⟨*Edge already present*⟩;
38: **end if**
39: **end procedure**

Algorithm 4 `searchLocal`(*cacheVer,* AU_{id}): Thread searches source node in *cache-List*.

40: **procedure** `searchLocal`($cacheVer$)
41: **if** (cacheVer.*inCnt*.CAS(0, -1)) **then**
42: *nCount* ← *nCount*.$get\&Inc()$;
43: AU_{id} ← cacheVer.AU_{id};
44: return⟨cacheVer⟩;
45: **else**
46: return⟨nil⟩;
47: **end if**
48: **end procedure**

Algorithm 5 `searchGlobal`(*BG,* AU_{id}): Thread searches the source node in BG.

49: **procedure** `searchGlobal`(*BG,* AU_{id})
50: *vNode* ← BG.*vHead*;
51: **while** (*vNode*.*vNext* ≠ BG.*vTail*) **do**
52: **if** (*vNode*.*inCnt*.CAS(0, -1)) **then**
53: *nCount* ← *nCount*.$get\&Inc()$;
54: AU_{id} ← *vNode*.AU_{id};
55: return⟨*vNode*⟩;
56: **end if**
57: *vNode* ← *vNode*.*vNext*;
58: **end while**
59: return⟨nil⟩;
60: **end procedure**

Algorithm 6 *decInCount(remNode)*: Decrement the *inCnt* of each conflicting node.

```
61: procedure decInCount(remNode)
62:     while (remNode.eNext ≠ remNode.eTail) do
63:         Decrement the inCnt atomically of
            remNode.vref in the vList;
64:         if (remNode.vref.inCnt == 0) then
65:             Add remNode.verf node into cacheList of
                thread local log, thLog;
66:         end if
67:         remNode ← remNode.eNext.verf;
68:         return⟨remNode⟩;
69:     end while
70:     return⟨nil⟩;
71: end procedure
```

Algorithm 7 *executeCode(curAU)*: Execute the current atomic-units.

```
72: procedure executeCode(curAU)
73:     while (curAU.steps.hasNext()) do /*Assume that
        curAU is a list of steps*/
74:         curStep = currAU.steps.next();
75:         switch (curStep) do
76:         case read(x):
77:             Read data-object x from a shared memory;
78:         case write(x, v):
79:             Write x in shared memory with value v;
80:         case default:
81:             /*Neither read or write in shared memory*/;
82:             execute curStep;
83:     end while
84:     return ⟨void⟩
85: end procedure
```

After successfully adding *vNode* in the BG, concurrent miner calls `addEdge()` method to add the conflicting node (or *eNode*) corresponding to *vNode* in the *eList*. First, the `addEdge()` method identifies the correct location of *eNode* in the *eList* of corresponding *vNode* at Line 28. If *eNode* is not part of *eList*, it creates and adds it into *eList* of *vNode* at Line 31 in a lock-free manner using atomic CAS operation. After successful addition of *eNode* in the *eList* of *vNode*, it increments the *inCnt* of *eNode*.$vref$ (to maintain indegree count) node, which is present in the *vList* at Line 32.

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 exists in the local *cacheList* with *inCnt* 0, then to claim that node, it sets the *inCnt* field to -1 at Line 41 atomically.

If the source node does not exist in the local *cacheList*, then the concurrent validator thread calls `searchGlobal()` method to identify the source node in the BG at Line 52. If a source node exists in the BG, it sets *inCnt* to -1 atomically to claim

Algorithm 8 *Concurrent Miner(*auList[]*, STM)*: Concurrently m threads are executing atomic-units from *auList[]* (or list of atomic-units) with the help of STM.

```
86: procedure Concurrent Miner(auList[], STM)
87:     /*Add all AUs in the Concurrent Bin (concBin[])*/
88:     concBin[] ← auList[];
89:     /*curAU is the current AU taken from auList[] */
90:     curAU ← curInd.get&Inc(auList[]);
91:     /*Execute until all AUs successfully completed*/
92:     while (curAU < size_of(auList[])) do
93:         T_i ← STM.begin();
94:         while (curAU.steps.hasNext()) do
95:             curStep = currAU.steps.next();
96:             switch (curStep) do
97:             case read(x):
98:                 v ← STM.read_i(x);
99:                 if (v == abort) then
100:                    goto Line 93;
101:                end if
102:            case write(x, v):
103:                STM.write_i(x, v);
104:            case default:
105:                /*Neither read or write in memory*/
106:                execute curStep;
107:        end while
108:        /*Try to commit the current transaction T_i and
            update the confList[i]*/
109:        v ← STM.tryC_i();
110:        if (v == abort) then
111:            goto Line 93;
112:        end if
113:        if (confList[i] == nil) then
114:            curAU doesn't have dependencies with other
                AUs. So, no need to create a node in BG.
115:        else
116:            create a nodes with respective dependencies
                from curAU to all AUs ∈ confList[i] in BG
                and remove curAU and AUs from concBin[]
117:            Create vNode with ⟨i, AU_id, 0, nil, nil⟩ as
                a vertex of Block Graph;
118:            BG(vNode, STM);
119:        end if
120:        curAU ← curInd.get&Inc(auList[]);
121:    end while
122: end procedure
```

that node and calls the `decInCount()` method to decreases the *inCnt* of all conflicting nodes atomically, which are present in the *eList* of corresponding source node at Line 63. While decrementing *inCnt*s, it 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 at Line 65.

4.2. Concurrent Miner

Smart contracts in blockchain are executed in two different contexts. First, the miner proposes a new block. Second, multiple validators re-execute to verify and validate the block proposed by the miner. In this subsection, we describe how miner executes the smart contracts concurrently.

A *concurrent miner* gets the set of transactions from the blockchain network. Each transaction is associated with a method (atomic-unit) of smart contracts. To run the smart contracts concurrently, we have faced the challenge of identifying the conflicting

transactions at run-time because smart contract languages are Turing-complete. Two transactionsconflict if they access a shared data-objects and at least one of them perform write operation. In *concurrent miner*, conflicts are identified at run-time using an efficient framework provided by the optimistic software transactional memory system (STMs). STMs access the shared data-objects called as *t-object*s. Each shared *t-object* is initialized to an initial state (or IS). The atomic-units may modify the IS to some other valid state. Eventually, it reaches the final state (or FS) at the end of block-creation. As shown in Algorithm 8, the concurrent miner first copies all the AUs in the concurrent bin at Line 88. Each transaction T_i gets the unique timestamp i from `STM.begin()` at Line 93. Then transaction T_i executes the atomic-unit of smart contracts. *Atomic-unit* consists of multiple steps such as $reads$ and $writes$ 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 97, 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, it gets the value as v at Line 98 and executes the next step of atomic-unit; otherwise, re-executed the atomic-unit if $aborted$ at Line 99.

If curStep is $write(x)$ at Line 102 then it calls the `STM.write(x)`. Internally, `STM.write()` stores the information of shared *t-object* x into local log (or *txlog*) in write-set (or $wset_i$) for transaction T_i. We use an optimistic approach in which the transaction's effect 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 are consistent, then it updates the TM; otherwise, re-execute the atomic-unit if $aborted$ at Line 110. After successful validation of `STM.tryC()`, it also maintains the conflicting transaction of T_i into the conflict list in TM.

If the conflict list is *nil* (Line 113), there is no need to create a node in the BG. Otherwise, create the node with respective dependencies in the BG and remove those AUs from the concurrent bin (Line 116). To maintain the BG, it calls `addVert()` and `addEdge()` methods of the lock-free graph library. The details of `addVert()` and `addEdge()` methods are explained in SubSection 4.1. Once the transactions successfully executed the atomic-units and done with BG construction, the *concurrent*

Algorithm 9 *Concurrent Validator*(*auList[]*, *BG)*: Concurrently V threads are executing AUs with the help of concurrent bin followed by the BG given by the miner.

```
123: procedure Concurrent Validator(auList[], BG)
124:     /*Execute until all AUs successfully completed*/
125:     /*Phase-1: Concurrent Bin AUs execution.*/
126:     while (concCount < size_of(concBin[])) do
127:         count ← concCount.get&Inc(auList[]);
128:         AU_id ← concBin[count];
129:         executeCode(AU_id);
130:     end while
131:     /*Phase-2: Block Graph AUs execution.*/
132:     while (nCount < size_of(auList[])) do
133:         while (cacheList.hasNext()) do
134:             cacheVer ← cacheList.next();
135:             cacheVertex ← searchLocal(cacheVer,
             AU_id);
136:             executeCode(AU_id);
137:             while (cacheVertex) do
138:                 cacheVertex ← decInCount(cacheVertex);
139:             end while
140:             Remove the current node (or cacheVertex)
             from local cacheList;
141:         end while
142:         vexNode ← searchGlobal(BG, AU_id);
143:         executeCode(AU_id);
144:         while (verNode) do
145:             verNode ← decInCount(verNode);
146:         end while
147:     end while
148: end procedure
```

miner computes the hash of the previous block. Eventually, *concurrent miner* proposes a block consisting of a set of transactions, BG, the final state of each shared *t-object*s, previous block hash, and sends it to all other network peers to validate.

4.3. Concurrent Validator

The concurrent validator validates the block proposed by the concurrent miner. It executes the block transactions concurrently and deterministically in two phases using a concurrent bin and BG given by the *concurrent miner*. In the first phase, validator threads execute the independent AUs of concurrent bin concurrently (Line 126 to Line 130). Then in the second phase, it uses BG to executes the dependent AUs by `executeCode()` method at Line 136 and Line 143 using `searchLocal()`, `searchGlobal()` and `decInCount()` methods of lock-free graph library at Line 135, Line 142 and (Line 138, Line 145), respectively. BG consists of dependency among the conflicting transactions that restrict them to execute serially. The functionality of lock-free graph library methods is explained earlier in SubSection 4.1.

After the successful execution of all the atomic-units, the *concurrent validator* compares its computed final state 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, based on consensus between network peers, the block is appended to the blockchain, and the respective *concurrent miner* is rewarded.

4.4. Optimizations

To make the proposed approach storage optimal and efficient, this subsection explains the key change performed on top of the solution proposed by Anjana et al. [1].

In Anjana et al. [1], there is a corresponding vertex node in the block graph (BG) for every AUs in the block. We observed that all the AUs in the block need not have dependencies. Adding a vertex node for such AUs takes additional space in the block. This is the first optimization our approach provides. In our approach, only the dependent AUs have a vertex in the BG, while the independent AUs are stored in the concurrent bin, which does not need any additional space. During the execution, a concurrent miner thread does not add a vertex to the BG if it identifies that the currently executed AU does not depend on the AUs already executed. However, suppose any other miner thread detects any dependence during the remaining AUs execution. That thread will add the dependent AUs vertices in the BG.

For example, let say we have n AUs in a block and a vertex node size is $\approx m$ kb to store in the BG, then it needs a total of $n * m$ kb of vertex node space for Anjana et al. [1]. Suppose from n AUs, only $\frac{n}{2}$ have the dependencies, then a total of $\frac{n}{2} * m$ kb vertex space needed in the BG. In the proposed approach, the space optimization can be 100% in the best case when all the AUs are independent. While in the worst case, it can be 0% when all the AUs are dependent. However, only a few AUs in a block have dependencies. Space-optimized BG helps to improve the network bandwidth and reduces network congestion.

Further, our approach combines the benefit of both Speculative Bin-based approach [11] and STM-based approach [1] to yield maximum speedup that can be achieved by validators to execute AUs. So, another optimization is at the validators side; due to the concurrent bin in the block, the time taken to traverse the BG will decrease; hence, speedup increases. The concurrent validators execution is modified and divided into two phases. First, it concurrently executes AUs of the concurrent bin using multiple threads, since AUs in the concurrent bin will be independent. While in the second

phase, dependent AUs are stored in the BG and concurrently executed using BG to preserve the transaction execution order as executed by the miner.

5. Correctness

The correctness of concurrent BG, miner, and validator is described in this section. We first list the linearization points (LPs) of the block graph library methods as follows:

1. `addVert`(*vNode*): (*vPred*.*vNext*.CAS(*vCurr*, *vNode*)) in Line 19 is the LP point of `addVert`() method if *vNode* is not exist in the BG. If *vNode* is exist in the BG then (*vCurr*.$ts_i \neq$ *vNode*.ts_i) in Line 17 is the LP point.

2. `addEdge`(*fromNode, toNode)*: (*ePred*.*eNext*.CAS(*eCurr*, *eNode*)) in Line 31 is the LP point of `addEdge`() method if *eNode* is not exist in the BG. If *eNode* is exist in the BG then (*eCurr*.$ts_i \neq$ toNode.ts_i) in Line 29 is the LP point.

3. `searchLocal`(cacheVer, AU_{id}): (cacheVer.*inCnt*.CAS(0, -1)) in Line 41 is the LP point of `searchLocal`() method.

4. `searchGlobal`(BG, AU_{id}): (*vNode*.*inCnt*.CAS(0, -1)) in Line 52 is the LP point of `searchGlobal`() method.

5. `decInCount(remNode)` : Line 63 is the LP point of `decInCount()` method.

Theorem 1. *Any history H_m generated by the concurrent miner using the BTO protocol satisfies co-opacity.*

Proof: Concurrent miner executes AUs concurrently using BTO protocol and generate a concurrent history H_m. The underlying BTO protocol ensures the correctness of concurrent execution of H_m. The BTO protocol [9, Chap 4] proves that any history generated by it satisfies co-opacity [23]. So, implicitly BTO proves that the history H_m generated by concurrent miner using BTO satisfies co-opacity.

Theorem 2. *Any history H_m generated by the concurrent miner using the MVTO protocol satisfies opacity.*

Proof: Concurrent miner executes AUs concurrently using MVTO protocol and generate a concurrent history H_m. The underlying MVTO protocol ensures the correctness of concurrent execution of H_m. The MVTO protocol [10] proves that any history generated by it satisfies opacity [8]. So, implicitly MVTO proves that the history H_m generated by concurrent miner using MVTO satisfies opacity.

Theorem 3. *All the dependencies between the conflicting nodes are captured in BG.*

Proof: Dependencies between the conflicting nodes are captured in the BG using LP points of lock-free graph library methods defined above. Concurrent miner constructs the lock-free BG using BTO and MVTO protocol in SubSection 4.1. BG consists of vertices and edges, where each committed AU act as a vertex and edges (or dependencies) represents the conflicts of the respective STM protocol (BTO and MVTO). As we know, STM protocols BTO [9, Chap 4] and MVTO [10] used in this paper for the concurrent execution are correct, i.e., these protocols captures all the dependencies correctly between the conflicting nodes. Hence, all the dependencies between the conflicting nodes are captured in the BG.

Theorem 4. *A history H_m generated by the concurrent miner using BTO protocol and a history H_v generated by a concurrent validator are view equivalent.*

Proof: A concurrent miner executes the AUs of H_m concurrently using BTO protocol, captures the dependencies of H_m in the BG, and proposes a block B. Then it broadcasts the block B along with BG to concurrent validators to verify the block B. The concurrent validator applies the topological sort on the BG and obtained an equivalent serial schedule H_v. Since the BG constructed from H_m considers all the conflicts and H_v obtained from the topological sort on the BG. So, H_v is equivalent to H_m. Similarly, H_v also follows the *read from* relation of H_m. Hence, H_v is legal. Since H_v and H_m are equivalent to each other, and H_v is legal. So, H_m and H_v are view equivalent.

Theorem 5. *A history H_m generated by the concurrent miner using MVTO protocol and a history H_v generated by a concurrent validator are multi-version view equivalent.*

Proof: Similar to the proof of Theorem 4, the concurrent miner executes the AUs of H_m concurrently using MVTO protocol, captures the dependencies in the BG, proposes a block B, and broadcasts it to the concurrent validators to verify it. MVTO maintains multiple-version corresponding to each shared object. Later, concurrent validator obtained H_v by applying topological sort on the BG provided by the concurrent miner. Since, H_v obtained from topological sort on the BG so, H_v is equivalent to H_m. Similarly, the BG maintains the *read from* relations of H_m. So, from MVTO protocol if T_j reads a value for shared object k say $r_j(k)$ from T_i in H_m then T_i committed before $r_j(k)$ in H_v. Therefore, H_v is valid. Since H_v and H_m are equivalent to each other and H_v is valid. So, H_m and H_v are multi-version view equivalent.

6. Experimental Evaluation

We aim to increase the efficiency of the miners and validators by employing concurrent execution of AUs while optimizing the size of the BG appended by the miner in the block. To assess the efficiency of the proposed approach, we performed simulation on the series of benchmark experiments with Ethereum [4] smart contracts from Solidity documentation [6]. Since multi-threading is not supported by the Ethereum Virtual Machine (EVM) [4, 5], we converted the Ethereum smart contracts into C++. We evaluated the proposed approach with the state-of-the-art approaches [1, 5, 11] over baseline serial execution on three different workloads by varying the number of AUs, the number of threads, and the number of shared objects. The benchmark experiments are conservative and consist of one or fewer smart contracts AUs in a block, which leads to a higher degree of conflicts than actual conflicts in practice where a block consists of AUs from different contracts ($\approx$ 1.5 million deployed smart contracts [24]). Due to fewer conflicts in the actual blockchain, the proposed approach is expected to provide greater concurrency. We structure our experimental evaluation to answer the following questions:

1. How much speedup is achieved with varying AUs by concurrent miners and validators when fixing the number of threads and shared objects? As conflicts increase with increasing AUs, we expect a decrease in speedup.

2. How does speedup change when increasing the number of threads with a fixed number of AUs and shared objects? We expect to see the speedup increase with increasing threads confined by logical threads available within the system.

3. How does speedup shift over different shared objects with fixed AUs and threads? We expect an increase in speedup due to conflict deterioration with objects increase. So, we anticipate concurrent miners and validators overweigh serial miners and validators with fewer conflicts.

6.1. Contract Selection and Benchmarking

This section provides a comprehensive overview of benchmark contracts coin, ballot, and simple auction from Solidity Documentation [6] selected as real-world examples for evaluating the proposed approach. The AUs in a block for the coin, ballot, and auction benchmark operate on the same contract, i.e., consists of the transaction calls of one or more methods of the same contract. In practice, a block consists of the AUs from different contracts; hence we designed another benchmark contract called *mix contract* consisting of contract transactions from coin, ballot, and auction in equal proportion in a block. The benchmark contracts and respective methods are as follows:
Coin Contract: The coin contract is the simplest form of sub-currency. The users involved in the contract have accounts, and *accounts* are shared objects. It implements methods such as `mint()`, `transfer()/send()`, and `getbalance()` which represent the AUs in a block. The contract deployer uses the `mint()` method to give initial coins/balance to each account with the same fixed amount. We initialized the coin contract's initial state with a fixed number of accounts on all benchmarks and workloads. Using `transfer()`, users can transfer coin from one account to other account. The `getbalance()` is used to check the coins in a user account. For the experiments a block consists of 75% `getbalance()`, and 25% `transfer()` calls. A conflict between AUs occurs if they access a common object (account), and at least one of them performs a `transfer()` operation.
Ballot Contract: The ballot contract is an electronic voting contract in which *voters* and *proposals* are shared objects. The `vote()`, `delegate()`, and `winningproposal()` are the methods of ballot contract. The voters use the `vote()` method to cast their vote

to a specific proposal. Alternatively, a voter can delegate their vote to other voter using `delegate()` method. A voter can cast or delegate their vote only once. At the end of the ballot, the `winningproposal()` is used to compute the winner. We initialized the ballot contract's initial state with a fixed number of proposals and voters for benchmarking on different workloads for experiments. The proposal to voter ratio is fixed to 5% to 95% of the total shared objects. A block consists of 90% `vote()`, and a 10% `delegate()` method calls followed by a `winningproposal()` call for the experiments. The AUs will conflict if they operate on the same object. So, if two voters `vote()` for the same proposal simultaneously, then they will conflict.

Simple Auction Contract: It is an online auction contract in which bidders bid for a commodity online. In the end, the amount from the maximum bidder is granted to the owner of the commodity. The *bidders*, *maximum bid*, and *maximum bidder* are the shared object. In our experiments, the initial contract state is a fixed number of bidders with a fixed initial account balance and a fixed period of the auction to end. In the beginning, the maximum bidder and bid are set to null (the base price and the owner can be set accordingly). The bidder uses the contract method `bid()` to bid for the commodity with their bid amount—the max bid amount and the bidder changes when a bid is higher than the current maximum. A bidder uses the `withdraw()` method to move the balance of their previous bid into their account. The bidder uses `bidEnd()` method to know if the auction is over. Finally, when the auction is ended, the maximum bidder (winner) amount is transferred to the commodity owner, and commodity ownership is transferred to the max bidder. For benchmarking in our experiments a block consist of 8% `bid()`, 90% `withdraw()`, and 2% `bidEnd()` method calls. The max bidder and max bid are the conflict points whenever a new bid with the current highest amount occurs.

Mix Contract: In this contract, we combine the AUs in equal proportion from the above three contracts (coin, ballot, and auction). Therefore, our experiment block consists of an equal number of corresponding contract transactions with the same initial state as initialized in the above contracts.

6.2. Experimental Setup and Workloads

We ran our experiments on a large-scale 2-socket Intel(R) Xeon(R) CPU E5-2690 V4 @ 2.60 GHz with a total of 56 hyper-threads (14 cores per socket and two threads per core) with 32 GB of RAM running Ubuntu 18.04.

In our experiments, we have noticed that speedup varies from contract to contract on different workloads. The speedup on various contracts is not for comparison between contracts. Instead, it demonstrates the proposed approach efficiency on several use-cases in the blockchain. We have considered the following three workloads for performance evaluation:

1. In workload 1 (W1), a block consists of AUs varies from 50 to 400, fixed 50 threads, and shared objects of 2K. The AUs per block in Ethereum blockchain is on an average of 100, while the actual could be more than 200 [5], however a theoretical maximum of ≈ 400 [25] after a recent increase in the gas limit. Over time, the number of AUs per block is increasing. In practice, one block can have less AUs than the theoretical cap, which depends on the gas limit of the block and the gas price of the transactions. We will see that in a block, the percentage of data conflicts increase with increasing AUs. The conflict within a block is described by different AUs accessing a common shared object, and at least one of them performs an update. We have found that the data conflict varies from contract to contract and has a varied effect on speedup.

2. In workload 2 (W2), we varied the number of threads from 10 to 60 while fixed the AUs to 300 and shared objects to 2K. Our experiment system consists of a maximum of 56 hardware threads, so we experimented with a maximum of 60 threads. We observed that the speedup of the proposed approach increases with an increasing number of threads limited by logical threads.

3. The number of AUs and threads in workload 3 (W3) are 50 and 300, respectively, although the shared objects range from 1K to 6K. This workload is used with each contract to measure the impact of the number of participants involved. Data conflicts are expected to decrease with an increasing number of shared objects;

however, the search time may increases. The speedup depends on the execution of the contract; but, it increases with an increasing number of shared objects.

6.3. Analysis

In our experiments, blocks of AUs were generated for each benchmark contract on three workloads: W1 (varying AUs), W2 (varying threads), and W3 (varying shared objects). Then, concurrent miners and validators execute the blocks concurrently. The corresponding blocks serial execution is considered as a baseline to compute the speedup of proposed concurrent miners and validators. The running time is collected for 15 iterations (times) with 10 blocks per iteration, and 10 validators validate each block. The first block of each iteration is left as a warm-up run, and a total of 150 blocks are created for each reading. So, each block execution time is averaged by 9. Further, the total time taken by all iterations is averaged by the number of iteration for each reading; the Eqn(1) is used to compute a reading time.

$$\alpha_t = \frac{\sum_{i=1}^{n} \sum_{b=1}^{m-1} \beta_t}{n * (m-1)} \tag{1}$$

Where α_t is an average time for a reading, n is the number of iterations, m is the number of blocks, and β_t is block execution time.

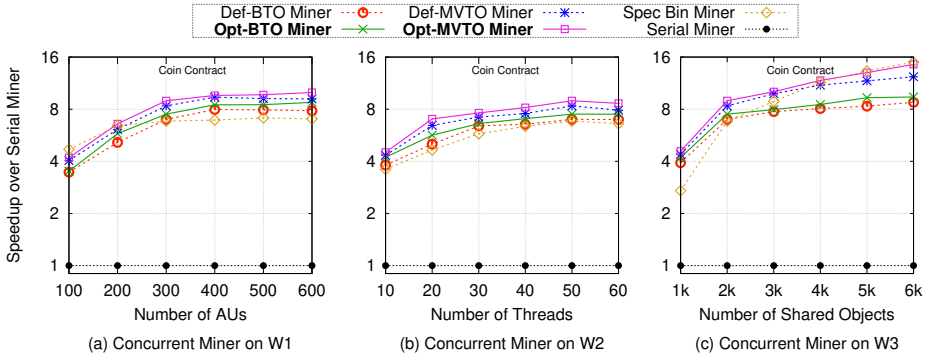
In all plots, figure (a), (b), and (c) correspond to workload W1, W2, and W3, respectively. Figure 3 to Figure 6 show the speedup achieved by proposed and state-of-the-art concurrent miners over serial miners for all benchmarks and workloads. Figure 7 to Figure 10 show the speedup achieved by proposed and state-of-the-art concurrent decentralized validators over serial validators for all benchmarks and workloads. Figure 11 to Figure 14 show speedup achieved by proposed and state-of-the-art concurrent fork-join validators over serial validators. Figure 15 to Figure 18 show the average number of edges (dependencies) and vertices (AUs) in the block graph for respective contracts on all workloads. While Figure 19 to Figure 22 show the percentage of additional space required to store the block graph in Ethereum block. A similar observation has been found [26] for the fork-join validator, the average number of dependencies, and space requirement on other contracts.

We observed that speedup for all benchmark contracts follows the roughly same pattern. In the read-intensive benchmarks (coin and mix), speedup likely to increase on all the workloads, while in the write-intensive benchmark (ballot and auction), speedup drop downs on high contention. We also observed that there might not be much speedup for concurrent miners with fewer AUs (less than 100) in the block, conceivably due to multi-threading overhead. However, the speedup for concurrent validators generally increases across all the benchmarks and workloads. Fork-join concurrent validators on W2 is an exception in which speedup drops down with an increase in the number of threads since fork-join follows a master-slave approach where a master thread becomes a performance bottleneck. We also observed that the concurrent validators achieve a higher speedup than the concurrent miners. Because the concurrent miner executes the AUs non-deterministically, finds conflicting AUs, creates concurrent bin and an efficient BG for the validators to execute the AUs deterministically.

Our experiment results also show the BG statics and additional space required to store BG in a block of Ethereum blockchain, which shows the space overhead. We compare our proposed approach with the existing speculative bin (Spec Bin) based approach [11], the fork-join approach (FJ-Validator) [5] and the approach proposed in [1] (we call it default/Def approach). The proposed approach combines the benefit of both bin-based and the STM approaches to get maximum benefit for concurrent miners and validators. The proposed approach[2] produces an optimal BG, reduces the space overhead, and outperforms the state-of-the-art approaches.

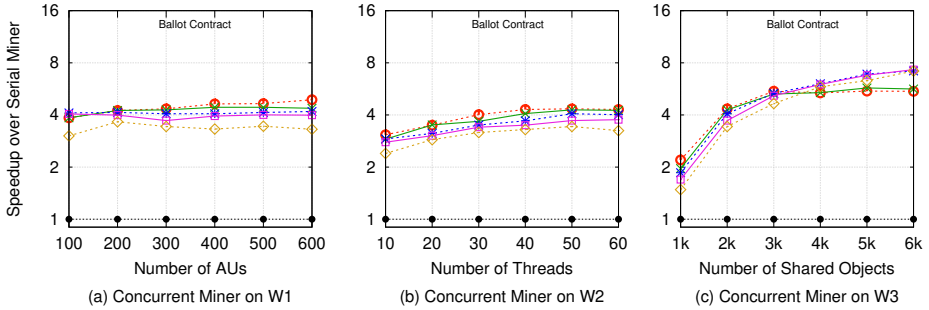
Figure 3(a) to Figure 6(a) show the speedup for concurrent miner on W1. As shown in Figure 3(a) and Figure 6(a) for read-intensive contracts such as in coin and mix contract, the speedup increases with an increase in AUs, respectively. While in write-intensive contracts such as ballot and auction contract the speedup does not increase with an increase in AUs; instead, it may drop down if AUs increases, as shown in Figure 4(a) and Figure 5(a), respectively. This is because contention increases with an increase in AUs.

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



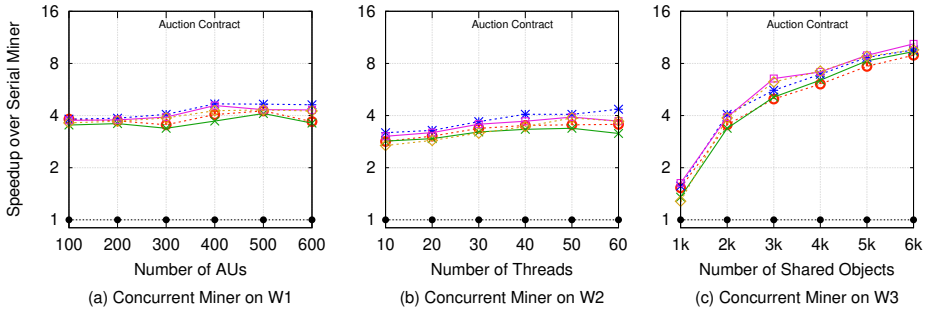
(a) Concurrent Miner on W1 (b) Concurrent Miner on W2 (c) Concurrent Miner on W3

Figure 3: Concurrent miner speedup over serial miner for coin contract.



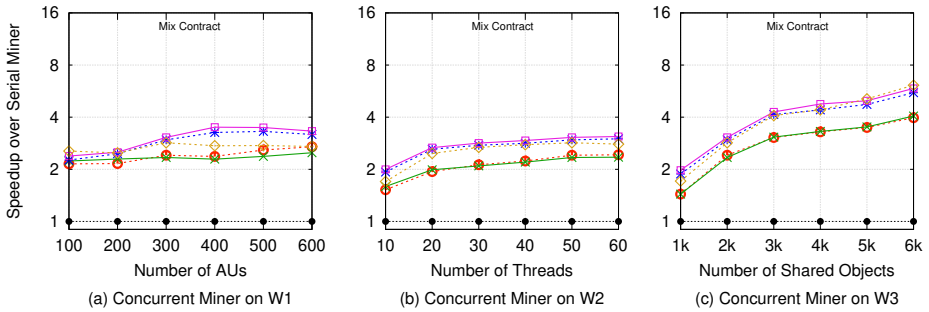
(a) Concurrent Miner on W1 (b) Concurrent Miner on W2 (c) Concurrent Miner on W3

Figure 4: Concurrent miner speedup over serial miner for ballot contract.



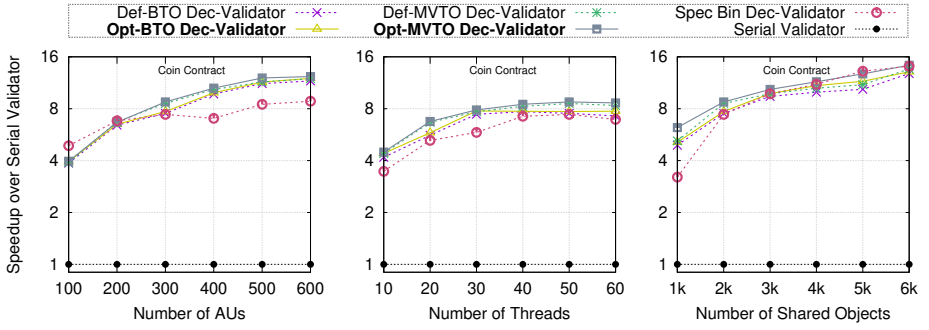
(a) Concurrent Miner on W1 (b) Concurrent Miner on W2 (c) Concurrent Miner on W3

Figure 5: Concurrent miner speedup over serial miner for auction contract.



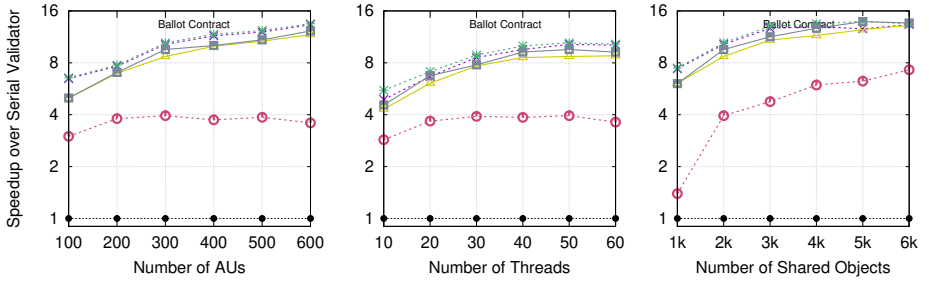
(a) Concurrent Miner on W1 (b) Concurrent Miner on W2 (c) Concurrent Miner on W3

Figure 6: Concurrent miner speedup over serial miner for mix contract.



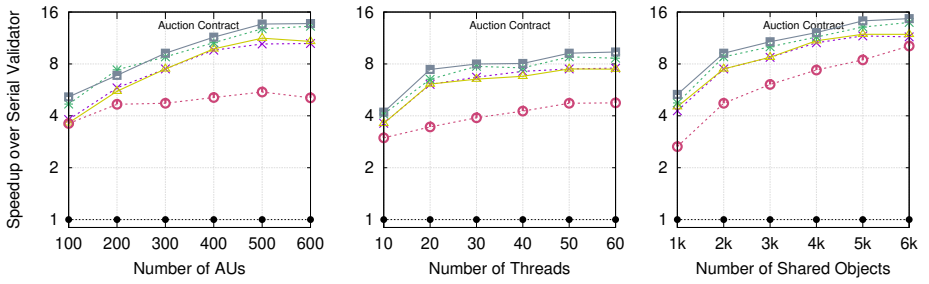
(a) Concurrent Dec-Validator on W1 (b) Concurrent Dec-Validator on W2 (c) Concurrent Dec-Validator on W3

Figure 7: Concurrent decentralized validator speedup over serial validator for coin contract.



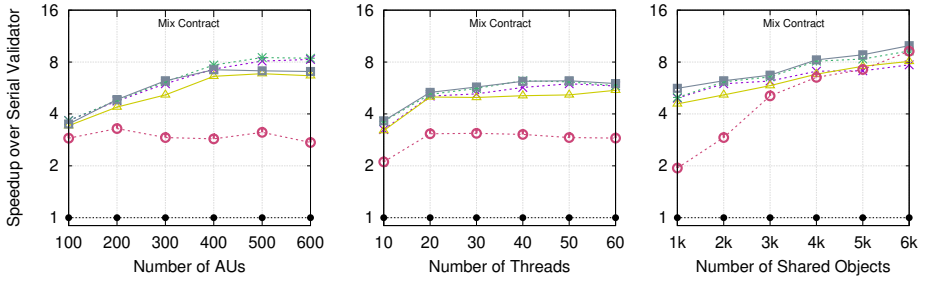
(a) Concurrent Dec-Validator on W1 (b) Concurrent Dec-Validator on W2 (c) Concurrent Dec-Validator on W3

Figure 8: Concurrent decentralized validator speedup over serial validator for ballot contract.



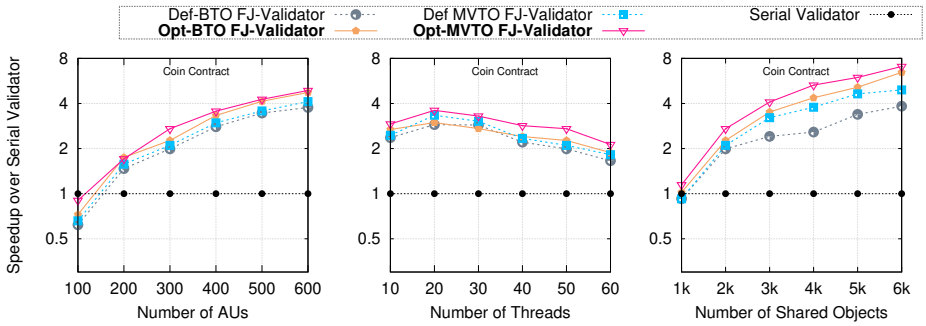
(a) Concurrent Dec-Validator on W1 (b) Concurrent Dec-Validator on W2 (c) Concurrent Dec-Validator on W3

Figure 9: Concurrent decentralized validator speedup over serial validator for auction contract.



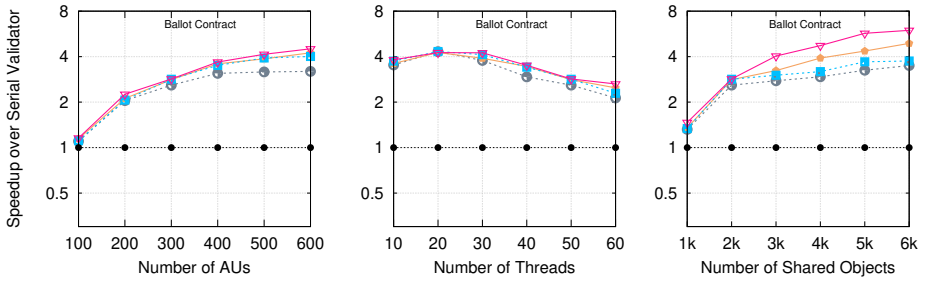
(a) Concurrent Dec-Validator on W1 (b) Concurrent Dec-Validator on W2 (c) Concurrent Dec-Validator on W3

Figure 10: Concurrent decentralized validator speedup over serial validator for mix contract.



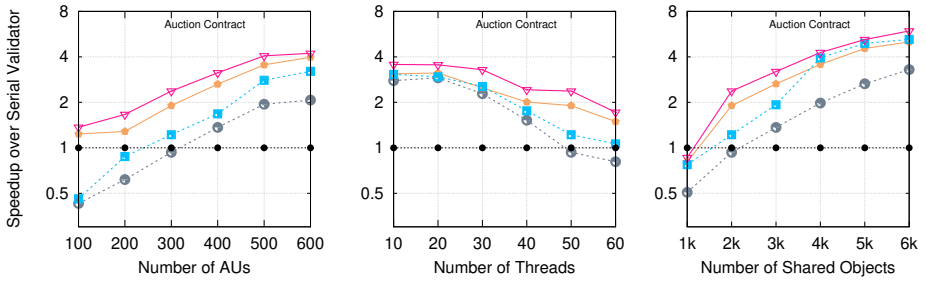
(a) Concurrent FJ-Validator on W1 (b) Concurrent FJ-Validator on W2 (c) Concurrent FJ-Validator on W3

Figure 11: Concurrent fork join validator speedup over serial validator for coin contract.



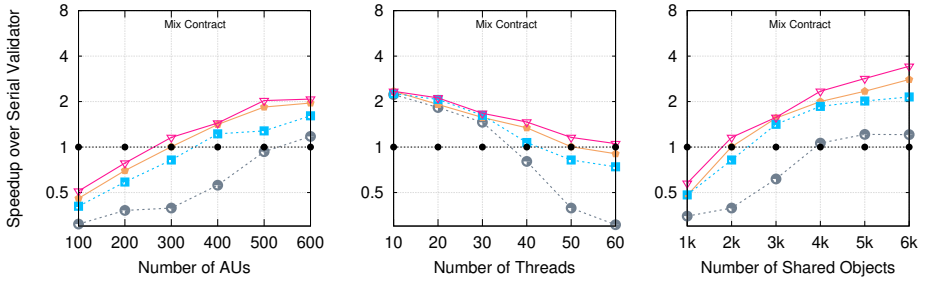
(a) Concurrent FJ-Validator on W1 (b) Concurrent FJ-Validator on W2 (c) Concurrent FJ-Validator on W3

Figure 12: Concurrent fork join validator speedup over serial validator for ballot contract.



(a) Concurrent FJ-Validator on W1 (b) Concurrent FJ-Validator on W2 (c) Concurrent FJ-Validator on W3

Figure 13: Concurrent fork join validator speedup over serial validator for auction contract.



(a) Concurrent FJ-Validator on W1 (b) Concurrent FJ-Validator on W2 (c) Concurrent FJ-Validator on W3

Figure 14: Concurrent fork join validator speedup over serial validator for mix contract.

Figure 7(a) through Figure 14(a) show the speedup for concurrent validators over serial validators on W1. The speedup for concurrent validators (decentralized and fork-join) increases with an increase in AUs. Figure 7(a) to Figure 10(a) demonstrate the speedup achieved by decentralized validator. It can be observed that for read-intensive benchmarks, the optimized MVTO decentralized validator (Opt-MVTO Dec-Validator) outperforms other validators. In contrast, in write-intensive benchmarks, the default MVTO decentralized validator (Def-MVTO Dec-Validator) achieves better speedup over other validators. Due to the overhead of multithreading for the concurrent bin with very fewer AUs. We observed that with increasing AUs in the blocks, the conflicts also increase. As a result, the number of transactions in the concurrent bin decreases. The speculative bin decentralized validator (Spec Bin Dec-Validator) speedup is quite less over concurrent STM Dec-Validators. Because STM miner precisely determines the dependencies between the AUs of the block and harness the maximum concurrency than the bin-based miner. However, suppose the block consists of the AUs with very few dependencies. In that case, Spec Bin Dec-Validator is expected to outperform other validators, as shown in the Figure 7(a).

Figure 11(a) to Figure 14(a) show the speedup for fork-join validators on W1 for all the benchmarks. We can observe that the proposed optimized MVTO fork-join validator (Opt-MVTO FJ-Validator) outperforms other validators due to lower overheads at the fork-join master validator thread to allocate independent AUs to slave validator threads. We noticed that decentralized concurrent validators speedup is quite high over fork-join concurrent validators because there is no bottleneck in this approach for allocating the AUs. All threads in the decentralized approach work independently. It can also be observed that with fewer AUs in several benchmarks, the speedup by fork-join validators drops to the point where it is less than the serial validators due to the overhead of thread creation dominate the speedup achieved, as shown in Figure 12(a), Figure 13(a) and Figure 14(a).

In W1, concurrent miners achieve a minimum of $\approx 2\times$ and maximum up to $10\times$ speedup over serial miners across the contracts. The concurrent STM decentralized validators achieve speedup minimum $\approx 4\times$ and maximum up to $\approx 14\times$ while Spec Bin Dec-Validator ranges from $\approx 3\times$ to $\approx 9\times$ over serial miner across the contracts.

The fork-join concurrent validators achieve a maximum speedup of $\approx 5\times$ over the serial validator.

Figure 3(b) to Figure 14(b) show the speedup on W2. The speedup increases with an increase in the number of threads. However, it is limited by the maximum number of logical threads in the experimental system. Thus, a slight drop in the speedup can be seen from 50 threads to 60 threads because the experimental system has a maximum of 56 logical threads. The reset of the concurrent miner observations is similar to the workload W1 based on read-intensive and write-intensive benchmarks.

As shown in the Figure 7(b) to Figure 10(b), the concurrent decentralized validators speedup increase with an increase in threads. While as shown in Figure 11(b) to Figure 14(b), the concurrent fork-join validators speedup drops down with an increase in threads. The reason for this drop in the speedup is that the master validator thread in the fork-join approach becomes a bottleneck. The decentralized validator's observation shows that for the read-intensive benchmark, the Opt-MVTO Dec-validator outperforms other validators. While in the write-intensive benchmark, the Def-MVTO Dec-validator outperforms other validators, as shown in Figure 8(b). However, in the fork-join validator approach, the proposed Opt-MVTO FJ-validator outperforms all other validators due to the optimization benefit of bin based approach inclusion.

In W2, concurrent miners achieve a minimum of $\approx 1.5\times$ and achieves maximum up to $\approx 8\times$ speedup over serial miners across the contracts. The concurrent STM decentralized validators achieve speedup minimum $\approx 4\times$ and maximum up to $\approx 10\times$ while Spec Bin Dec-Validator ranges from $\approx 3\times$ to $\approx 7\times$ over serial miner across the contracts. The fork-join concurrent validators achieve a maximum speedup of $\approx 4.5\times$ over the serial validator.

The plots in Figure 3(c) to Figure 14(c) show the concurrent miners and validators speedup on W3. As shared objects increase, the concurrent miner speedup increases because conflict decreases due to less contention. Additionally, when contention is very low, more AUs are added in the concurrent bin. However, it also depends on the contract. If the contract is a write-intensive, fewer AUs are added in the concurrent bin. While more AUs added in the concurrent bin for read-intensive contracts.

As shown in Figure 3(c) and Figure 6(c), the speculative bin miners surpass STM

miners due to read-intensive contracts. While in Figure 4(c) and Figure 5(c), the Def-MVTO Miner outperform other miners as shared objects increase. In contrast, Def-BTO Miner performs better over other miners when AUs are fewer because search time in write-intensive contracts to determine respective versions is much more in MVTO miner than BTO miner. Although, all concurrent miners performers better than the serial miner. In W3, concurrent miners start at around $1.3\times$ and archives maximum up to $14\times$ speedup over serial miners across all the contracts.

The speedup by validators (decentralized and fork-join) increases with shared objects. In Figure 7(c), Figure 9(c), and Figure 10(c), proposed Opt-STM Dec-Validator perform better over other validators. However, for write-intensive contracts, the number of AUs in the concurrent bin would be less. Therefore, the speedup by Def-STM Dec-Validators is greater than Opt-STM Dec-Validators, as shown in Figure 8(c). The Spec Bin Dec-Validator speedup is quite less over concurrent STM Dec-Validators because STM miner precisely determines the dependencies between the AUs than the bin based miner.

In fork-join validators, proposed Opt-STM FJ-Validators outperform over all other FJ-Validators, as shown in Figure 11(c) to Figure 14(c) because of less contention at the master validator thread in the proposed approach to allocate independent AUs to slave validator threads. We noticed that decentralized concurrent validators speedup is relatively high over fork-join concurrent validators with similar reasoning explained above. In W3, concurrent STM decentralized validators start at around $4\times$ and achieve a maximum up to $14\times$ speedup while Spec Bin Dec-Validator ranges from $1\times$ to $14\times$ speedup over serial miner across the contracts. The fork-join concurrent validators achieve a maximum speedup of $7\times$ over the serial validator. The concurrent validators benefited from the work of the concurrent miners and outperformed serial validators.

Figure 15 to Figure 18 show the average number of edges (dependencies as histograms) and vertices (AUs as line chart) in the BG for mix contract on all the workloads[3]. The average number of edges (dependencies) in the BG for both Default and

[3]We used histograms and line chart to differentiate vertices and edges to avoid confusion in comparing the edges and vertices.

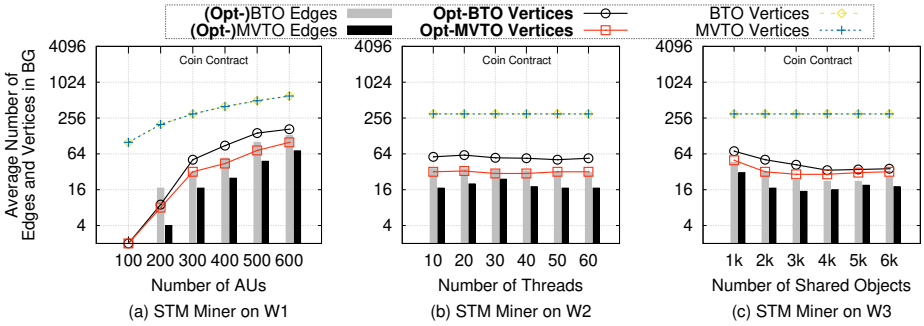


Figure 15: Average number of edges (dependencies) and vertices (AUs) in block graph for coin contract.

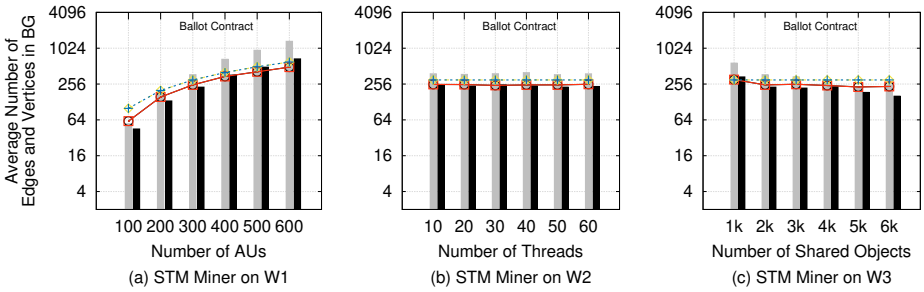


Figure 16: Average number of edges (dependencies) and vertices (AUs) in block graph for ballot contract.

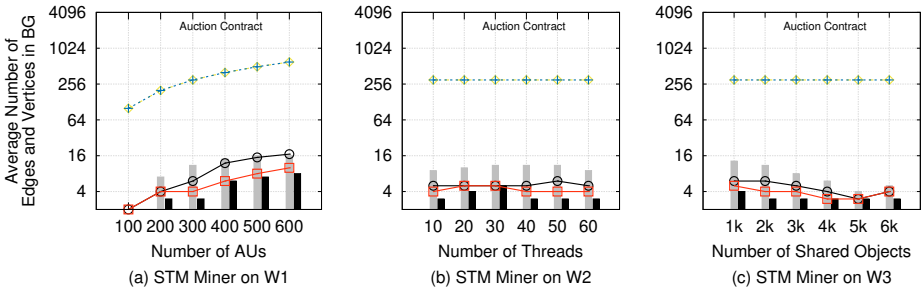


Figure 17: Average number of edges (dependencies) and vertices (AUs) in block graph for auction contract.

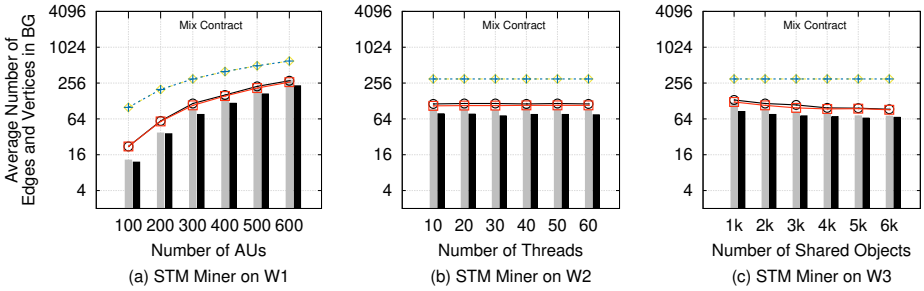


Figure 18: Average number of edges (dependencies) and vertices (AUs) in block graph for mix contract.

Optimized approach for respective STM protocol remains the same; hence only two histograms are plotted for simplicity. As shown in the Figure 15(a) to Figure 18(a) with increasing AUs in W1, the BG edges and vertices also increase. It shows that the contention increases with increasing AUs in the blocks. As shown in the Figure 15(b) to Figure 18(b) in W2, the number of vertices and edges does not change much. However, in the W3, the number of vertices and edges decreases, as shown in Figure 15(c) to Figure 18(c).

In our proposed approach, the BG consists of vertices respective to only conflicting AUs, and non-conflicting AUs are stored in the concurrent bin. While in Anjana et al. [1] approach, all the AUs had corresponding vertex nodes in the BG shown in Figure 15 to Figure 18. So, in W1, it will be 100 vertices in the BG if block consists of 100 AUs and 200 if block consists of 200 AUs. In W2 and W3, it will be 300 vertices. Having only conflicting AUs vertices in BG saves much space because each vertex node takes 28-byte storage space.

The average block size in the Bitcoin and Ethereum blockchain is ≈ 1200 KB [27] and ≈ 20.98 KB [28], respectively measured for the interval of Jan 1^{st}, 2019 to Dec 31^{th}, 2020. Further, the block size keeps on increasing, and so the number of transactions in the block. The average number of transactions in the Ethereum block is ≈ 100 [28]. Therefore, in the Ethereum blockchain, each transaction size is an average ≈ 0.2 KB (≈ 200 bytes). We computed the block size based on these simple calculations when AUs vary in the block for W1. The Eqn(2) is used to compute the block size (B) for the experiments.

$$B = 200 * N_{AUs} \tag{2}$$

Where, B is block size in bytes, N_{AUs} number of AUs in block, and 200 is the average size of an AU in bytes.

To store the block graph $BG(V, E)$ in the block, we used *adjacency list*. In the BG, a vertex node V_s takes 28 bytes storage, which consists of 3 integer variables and 2 pointers. While an edge node E_s needs a total of 20 bytes storage. The Eqn(3) is used to compute the size of BG (β bytes). While Eqn(4) is used to compute the additional space (β_p percentage) needed to store BG in the block.

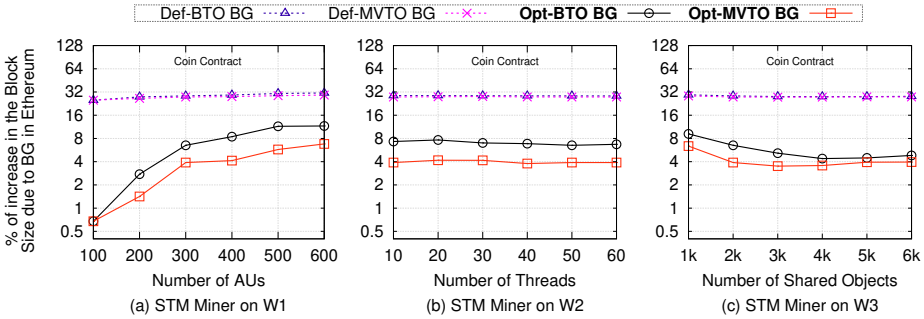


Figure 19: Percentage of additional space to store block graph in Ethereum block for coin contract.

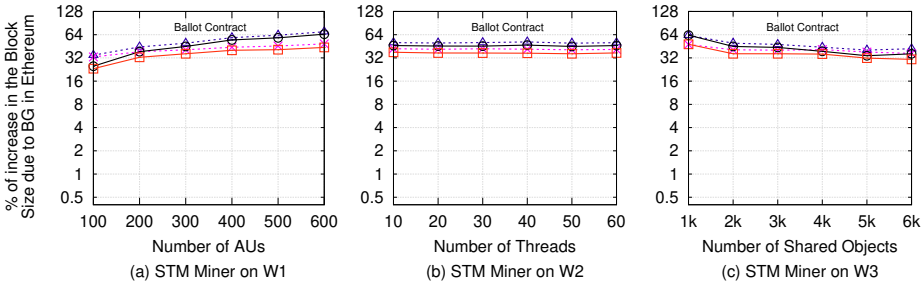


Figure 20: Percentage of additional space to store block graph in Ethereum block for ballot contract.

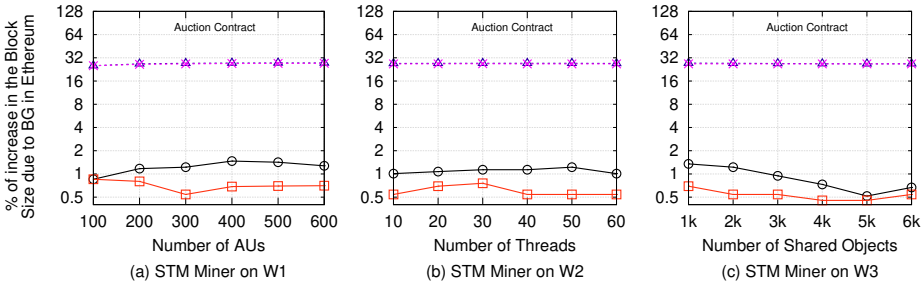


Figure 21: Percentage of additional space to store block graph in Ethereum block for auction contract.

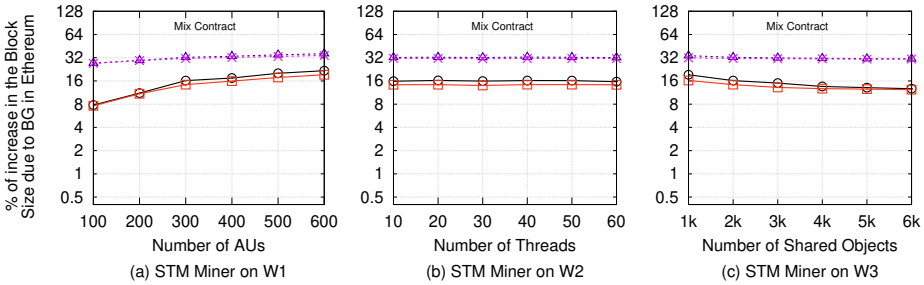


Figure 22: Percentage of additional space to store block graph in Ethereum block for mix contract.

$$\beta = (V_s * N_{AUs}) + (E_s * M_e) \tag{3}$$

Where, β is size of BG in bytes, V_s is size of a vertex node of BG in bytes, E_s is size (in bytes) of a edge node of BG, and M_e is number of edges in BG.

$$\beta_p = (\beta * 100)/B \tag{4}$$

The plots in Figure 19 to Figure 22 demonstrate the average percentage of additional storage space required to store BG in the Ethereum block on all benchmarks and workloads. We can observe that the space requirement also increases with an increase in the number of dependencies and vertices in BG. However, the space requirement of our proposed approach is smaller than the existing default approach. As shown in the Figure 16, the dependencies and vertices are highest in the ballot contract compared to other contracts, so the space requirement is also high, as shown in Figure 20. This is because the ballot is a write-intensive benchmark. It can be seen that the space requirements of BG by Opt-BTO BG and Opt-MVTO BG is smaller than Def-BTO BG and Def-MVTO BG miner, respectively.

The proposed approach significantly reduces the BG size for mix contract as shown in Figure 22 across all the workloads. Which clearly shows the storage efficiency of the proposed approach. The storage advantage comes from using a bin-based approach combined with the STM approach, where concurrent bin information needs to be added into the block, which requires less space than having a corresponding vertex in BG for each AUs of the block. So, we combine the advantages of both the approaches (STM and Bin) to get maximum speedup with storage optimal BG. The average space required for BG in % w.r.t. block size is 34.55%, 31.69%, 17.24%, and 13.79% by Def-BTO. Def- MVTO, Opt-BTO, and Opt-MVTO approach, respectively. The proposed Opt-BTO and Opt-MVTO BG are $2\times$ (or 200.47%) and $2.30\times$ (or 229.80%) efficient over Def-BTO and Def-MVTO BG, respectively. With an average speedup of $4.49\times$ and $5.21\times$ for Opt-BTO, Opt-MVTO concurrent miner over serial, respectively. The Opt-BTO and Opt-MVTO decentralized concurrent validator outperform an average of $7.68\times$ and $8.60\times$ than serial validator, respectively.

7. 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, the miner executes the smart contracts concurrently using optimistic STM protocol as BTO. To reduce the number of aborts and further improve efficiency, the concurrent miner uses MVTO protocol, which maintains multiple versions corresponding to each data object. Concurrent miner proposes a block that consists of a set of transactions, concurrent bin, BG, previous block hash, and the final state of each shared data objects. Later, the validators re-execute the same smart contract transactions concurrently and deterministically in two-phase using concurrent bin followed by the BG given by miner, which capture the conflicting relations among the transactions to verify the final state. Overall, the proposed Opt-BTO and Opt-MVTO BG are $2\times$ (or 200.47%) and $2.30\times$ (or 229.80%) efficient over Def-BTO and Def-MVTO BG, respectively. With an average speedup of $4.49\times$ and $5.21\times$ for Opt-BTO, Opt-MVTO concurrent miner over serial, respectively. The Opt-BTO and Opt-MVTO decentralized concurrent validator outperform an average of $7.68\times$ and $8.60\times$ than serial validator, respectively.

Acknowledgements. This project was partially supported by a research grant from Thynkblynk Technologies Pvt. Ltd, and MEITY project number 4(20)/2019-ITEA.

References

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

[2] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, A. Somani, Entitling concurrency to smart contracts using optimistic transactional memory, in: Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 508.

[3] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, https://bitcoin.org/bitcoin.pdf (2009).

[4] Ethereum, http://github.com/ethereum, [Acessed 26-3-2019].

[5] T. Dickerson, P. Gazzillo, M. Herlihy, E. Koskinen, Adding Concurrency to Smart Contracts, in: Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC '17, ACM, New York, NY, USA, 2017, pp. 303–312.

[6] Solidity Documentation, https://solidity.readthedocs.io/, [Accessed 15-09-2020].

[7] C. H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653.

[8] R. Guerraoui, M. Kapalka, On the correctness of transactional memory (2008) 175–184.

[9] G. Weikum, G. Vossen, Transactional Info Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery, 2002.

[10] P. Kumar, S. Peri, K. Vidyasankar, A TimeStamp Based Multi-version STM Algorithm, in: ICDCN, Springer, 2014, pp. 212–226.

[11] V. Saraph, M. Herlihy, An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts, in: International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019), Vol. 71 of OASIcs, Dagstuhl, Germany, 2020, pp. 4:1–4:15.

[12] M. Herlihy, N. Shavit, On the nature of progress, OPODIS 2011, Springer.

[13] N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (9).

[14] L. Luu, J. Teutsch, R. Kulkarni, P. Saxena, Demystifying incentives in the consensus computer, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, ACM, New York, NY, USA, 2015, pp. 706–719.

[15] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, E. 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., Springer.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: CCS '16, ACM.

[17] A. Zhang, K. Zhang, Enabling concurrency on smart contracts using multiversion ordering, in: Web and Big Data, Springer, Cham, 2018, pp. 425–439.

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

[19] P. S. Anjana, H. Attiya, S. Kumari, S. Peri, A. Somani, Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory, in: Networked Systems, Springer, Cham, 2021, pp. 77–93.

[20] R. Guerraoui, M. Kapalka, Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory, 2010.

[21] P. Kuznetsov, S. Peri, Non-interference and local correctness in transactional memory, Theor. Comput. Sci. 688 (2017) 103–116.

[22] M. P. Herlihy, J. M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst., 1990.

[23] S. Peri, A. Singh, A. Somani, Efficient means of achieving composability using object based semantics in transactional memory systems, in: Networked Systems, Springer, Cham, 2019, pp. 157–174.

[24] E. Muzzy, S. Rizvi, D. Sui, Ethereum by the numbers, `https://media.consensys.net/ethereum-by-the-numbers-3520f44565a9`, [Accessed 15-09-2020] (2018).

[25] N. Shukla, Ethereum's increased gas limit enables network to hit 44 tps, https://eng.ambcrypto.com/ethereums-increased-gas-limit-enables-network-to-hit-44-tps/amp/, [Accessed 15-09-2020].

[26] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, A. Somani, An efficient framework for concurrent execution of smart contracts, CoRR abs/1809.01326.
URL http://arxiv.org/abs/1809.01326

[27] Bitcoin Block Size, https://www.blockchain.com/en/charts, [Accessed 15-09-2020].

[28] Ethereum Stats, https://etherscan.io/charts, [Accessed 15-09-2020].