DAG-based Efficient Parallel Scheduler for Blockchains: Hyperledger Sawtooth as a Case Study*

Manaswini Piduguralla, Saheli Chakraborty,
Parwat Singh Anjana, and Sathya Peri

Indian Institute of Technology Hyderabad, India (502284)
{cs20resch11007,ai20mtech12002,cs17resch11004}@iith.ac.com,
sathya_p@cse.iith.ac.in.

Abstract. Blockchain technology is a distributed, decentralized, and immutable ledger system. It is the platform of choice for managing smart contract transactions (SCTs). Smart contracts are of pieces code that capture business agreement between interested parties and are commonly implemented using blockchains. A block in a blockchain contains a set of transactions representing changes to the system and a hash of the previous block. The SCTs are executed multiple times during the block production and validation phases across the network. In most of the existing blockchains, transactions are executed sequentially.
In this work, we propose a parallel direct acyclic graph (DAG) based scheduler module for concurrent execution of SCTs. This module can be seamlessly be integrated into the blockchain framework and the SCTs in a block can be executed efficiently resulting in higher throughput. The dependencies among the SCTs of a block are represented as DAG data structure which enables parallel execution of the SCTs. Furthermore, the DAG data structure is shared with block validators, allowing resource conservation for DAG creation across the network. To ensure secure parallel execution, we design a secure validator capable of validating and identifying incorrect DAGs shared by malicious block producers. For evaluation, our framework is implemented in Hyperledger Sawtooth V1.2.6. The performance across multiple smart contract applications is measured for the various schedulers. We observed that our proposed executor exhibits a 1.58 times performance improvement on average over serial execution.

Keywords: Smart Contract Executions, Blockchains, Hyperledger Sawtooth, Parallel Scheduler.

1 Introduction

Blockchain platforms help establish and maintain a decentralized and distributed ledger system between untrusting parties [15]. The blockchain is a collection of

* Funded by Meity India: No.4(4)/2021-ITEA & 4(20)/2019-ITEA. This is part of the National (Indian) Blockchain Framework Project.

immutable blocks, typically in the form of a chain. Each block points to its previous block by storing its hash. A block in the blockchain consists of several *smart contract transactions (SCTs)*, which are self-executing contracts of agreement between two or more parties that are written in the form of computer code. These help in the execution of agreements among untrusted parties without the necessity for a common trusted authority to oversee the execution. The development and deployment of smart contracts on blockchain platforms is growing rapidly.

A blockchain network usually consists of several nodes (ranging from thousands to millions depending on the blockchain), each of which stores the entire contents of the blockchain. Any node in the blockchain can act as a *block producer*. A producer node selects transactions from a pool of available transactions and packages them into a block. The proposed block is then broadcast to other nodes in the network. A node receiving the block acts as a *validator*. It validates the transactions in the block by executing them one after another. Thus a node can act as a producer while producing the block and as a validator for blocks produced by other nodes in the network.

Agreement on the proposed block by the nodes of the blockchain is performed through various consensus mechanisms, like proof of work (PoW) [15], proof of stake (PoS) [18], proof of elapsed time (PoET) [13], etc. For a block to be added to the blockchain, the transactions of the block are processed in two contexts: (a) first time by the block producer when the block is produced; (b) then by all the block validators as a part of the block validation. Thus the SCT code is executed multiple times by the producer and the validators.

The majority of blockchain technologies execute the SCTs in a block serially during the block creation and validation phases. This is one of the bottlenecks for higher throughput and scalability of blockchain models [10]. The throughput of the blockchain can be improved by concurrent execution of transactions. In order to enable concurrent transaction processing, it is crucial to ensure the avoidance of running interdependent transactions simultaneously. Moreover, when executing transactions concurrently at each validator, they must yield an identical end state in the database.

This work proposes a framework for executing transactions concurrently on producers and validators. We have implemented our framework in Hyperledger Sawtooth 1.2.6. [2]. We have chosen Sawtooth (explained in Section 2) as our platform of choice due to its existing support for parallel execution of SCTs, which provides us with an ideal environment to compare and test against both serial and parallel schedulers. This approach could be implemented in any blockchain with an order-execute blockchain model [4]. The major contributions of the paper are as follows:

- We introduced two important modules: a parallel scheduler and a secure validator are introduced in this work. The parallel scheduler module is responsible for identifying transaction dependencies within a block and scheduling them for conflict-free execution using a directed acyclic graph (DAG). The DAG is shared along with the block and validated by the secure validator

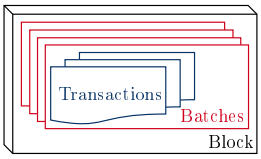

Figure. 1: Structure of a Hyperledger Sawtooth block.

module, which helps detect any malicious block producers. Section 3 provides a comprehensive explanation of the framework.
– We observed that our proposed executor achieves average speedups of 1.58 times and 1.29 times over Sawtooth's default serial executor and built-in parallel executor, respectively. The implementation details, experiment design, and results are discussed in Section 4.

The overview of the related work aligned with the proposed approach is discussed in Section 5, while the conclusion and future steps are discussed in Section 6.

2 Background on Hyperledger Sawtooth

The Hyperledger Foundation is an open-source collaboration project by the Linux Foundation to establish and encourage cross-industry blockchain technologies. Sawtooth is one of the most popular blockchain technologies being developed for permissioned and permissionless networks. It is designed such that transaction rules, permissions, and consensus algorithms can be customized according to the particular area of application. Some of the distinctive features of Sawtooth are modularity, multi-language support, parallel transaction execution, and pluggable consensus. The modular structure of Sawtooth helps in modifying particular operations without needing to make changes throughout the architecture.

In Sawtooth, smart contracts are referred to as *transaction families*, and the logic for the contract is present in the respective families' transaction processors. Modifications to the state are performed through transactions, and they are always wrapped inside a batch. A batch is the atomic unit of change in the system and multiple batches are combined to form a block (Figure 1). The node architecture of Sawtooth includes five modules that play crucial roles in blockchain development: global state, journal, transaction scheduler, transaction executor, and transaction processor. The global state containing the data of transaction families of Sawtooth is stored using a Merkle-Radix tree data structure. The Journal module contains a block completer, block validator, and block publishers that deal with creating, verifying, and publishing blocks. It is the responsibility of the Transaction Scheduler module to identify the dependencies between transactions and schedule transactions that result in conflict-free execution. In order to execute a transaction, the transaction executor collects the context of the transaction.[1]

[1] The detailed architecture is explained in Appendix A of [16].

Hyperledger Sawtooth architecture includes a parallel transaction scheduler (tree-scheduler) that uses a Merkle-Radix tree with nodes that are addressable by state addresses. This tree is called the predecessor tree. Each node in the tree represents one address in the Sawtooth state, and a read list and write list are maintained at each node. Whenever an executor thread requests the next transaction, the scheduler inspects the list of unscheduled transactions and the status of their predecessors. The drawbacks of the tree scheduler are that the status of predecessor transactions needs to be checked before a transaction can be scheduled. The construction of the tree data structure is serial. The number of addresses accessed in a block is generally higher than the total number of transactions. A data structure based on addresses typically requires more memory space compared to a transaction-based data structure. The block producers and validators both construct the tree at their end instead of the block producer sharing the tree with the validators.

The proposed framework for transaction execution on the blockchain would improve the throughput of SCTs by making the block creation and validation process concurrent. SCTs independent of each other are executed in parallel in the framework. The dependencies are represented as a DAG based on transaction inputs and outputs. DAG sharing and secure validator module designs are also included in the framework to further optimize block validation.

3 Proposed Framework

In this section, the proposed framework for parallel transaction execution in blockchains through static analysis of the block is detailed. This framework introduces *parallel scheduler* and *secure validator* modules into the blockchain node architecture, as shown in Figure 2. The parallel scheduler (SubSection 3.1) is responsible for identifying the dependencies among the transactions in the block and scheduling them for conflict-free execution. This is done by determining the dependencies among the transactions. The identified dependencies are represented by a DAG that is shared along with the block to minimize the validation time of the blockchain, the idea explored in [6, 7, 10]. DAG shared along with the blocks are received and validated by the *secure validator* (SubSection 3.2). Through the validation process, the secure validator determines if any malicious block producer has shared a graph with some crucial edge (dependency) missing. This section presents pseudo-codes for the modules as well as a detailed framework.

3.1 Parallel Scheduler

The parallel scheduler module is part of the block producer in the proposed framework. It performs the operations of DAG creation and conflict-free transaction execution. Both processes are multi-threaded for higher throughput.

DAG Creation: The DAG creation is initiated when the block producer creates the next block in the blockchain. In blockchains like Sawtooth, the addresses that

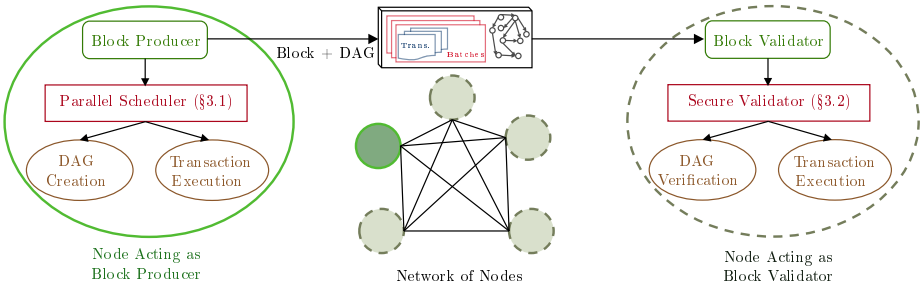

Figure. 2: Proposed framework in the blockchain

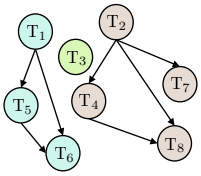

Figure. 3: DAG representation of dependencies in the block.

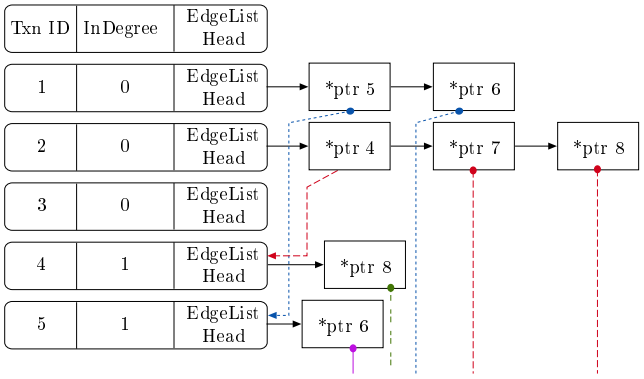

Figure. 4: Linked list representation of the DAG.

the transactions read from, and write to, are present in the transaction header. Using this information, we can derive the addresses based on the transaction details without having to execute the transaction. By examining the *input* (read) and *output* (write), the parallel scheduler calculates the dependencies among transactions, as described in the following explanation.

On receiving a block (from the block publisher module), the producer deploys multiple threads to generate the DAG. Firstly, a unique id is assigned to the transactions based on their order in the block ($T_1, T_2, T_3...$) using a global atomic counter as shown in the Algorithm 1, Line 3. The input addresses of the transaction (T_a) are compared with all the output addresses of transactions (e.g., T_b) with a higher ID. Correspondingly, the output addresses of T_a are compared with the input and output addresses of T_b as shown in Algorithm 1 from Line 10 to Line 24. If there are any common addresses identified in the above checks, an edge is added from T_a to T_b in the DAG. An adjacency matrix data structure is implemented for representing the graph, and an atomic array is used to maintain the indegree count of the vertices. We have also implemented a module with a concurrent linked list structure, as shown in Figure 4. The pseudo-code is detailed in Algorithm 1, and one can refer to appendix C in [16] for an in-depth explanation. We have also proved the safety of our proposed framework for concurrent execution, available in Appendix B in [16].

Algorithm 1 Multi-threaded createDAG(): m threads concurrently create DAG

```
1: procedure CREATEDAG(block)                ▷ The block to be produced or validated is the input
2:     while true do
3:         T_i ← txnCounter.get&inc()                          ▷ Claim the next transaction available
4:         if T_i > txn_count then
5:             txnCounter.dec()
6:             return                                  ▷ Return if all the transactions are processed
7:         end if
8:         Graph_Node *txn = new Graph_Node
9:         DAG→add_node(T_i, txn)                                        ▷ adding the node to the graph
10:        for T_j = T_i + 1 to txn_count do                          ▷ finding dependent transactions
11:            flagEdge=false
12:            if T_i.readList ∩ T_j.writeList then                ▷ Checking for RW and WW conflicts
13:                flagEdge=True
14:            end if
15:            if T_i.writeList ∩ T_j.readList then
16:                flagEdge=True
17:            end if
18:            if T_i.writeList ∩ T_j.writeList then
19:                flagEdge=True
20:            end if
21:            if flagEdge then
22:                DAG→add_edge(T_i, T_j)
23:            end if
24:        end for
25:    end while
26: end procedure                                  ▷ Threads join when the DAG is complete
```

Algorithm 2 Multi-threaded selectTxn(): threads concurrently search DAG for the next transaction to execute

```
27: procedure SELECTTXN(DAG)
28:     for T_i = pos To txn_count do   ▷ iterate over until all transactions to find transaction for
    execution
29:         if T_i.indeg == 0 then                                  ▷ Checking for txn with zero indegree
30:             if T_i.indeg.CAS(0, −1) then
31:                 pos ← T_i                                  ▷ store the last position for fast parsing
32:                 return T_i
33:             end if
34:         end if
35:     end for
36:     for T_i = 0 To pos do   ▷ iterate over until all transactions to find transaction for execution
37:         if T_i.indeg == 0 then                                  ▷ Checking for txn with zero indegree
38:             if T_i.indeg.CAS(0, −1) then
39:                 pos ← T_i                                  ▷ store the last position for fast parsing
40:                 return T_i
41:             end if
42:         end if
43:     end for
44:     return −1 ▷ Threads returns when a transaction is selected or all transactions are executed.
45: end procedure
```

Transaction Execution: Once the dependency DAG is constructed, the block producer proceeds to execute the transactions within the block in parallel. It initiates multiple threads to process the transactions. Each thread selects a transaction for execution using the indegree array like in Line 30 of Algorithm 2. If the indegree of a transaction is zero, it indicates that the transaction does not have any predecessor-dependent transactions and can be executed (T_1, T_3, and T_2 in Figure 3). If no such transactions are available, the threads wait until one is available or end execution if all the transactions have completed execution. Upon selecting a transaction, it is executed, and the indegrees of all the outgoing edge transactions (T_5 and T_6 for T_1) are decremented by 1. Then, the next transaction

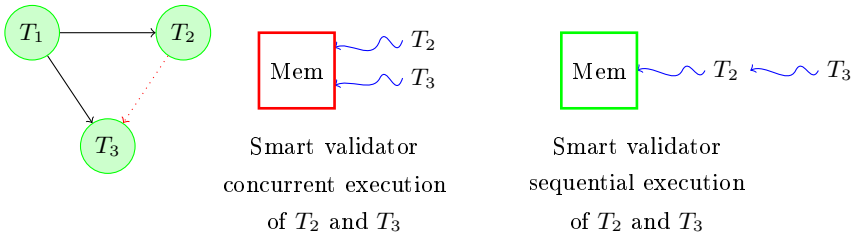

Figure. 5: Example scenario of smart validator proposed by Anjana et al. in [6]

with zero indegree is searched for. This search can be optimized by initiating the search from the last transaction ID selected. The last transaction ID selected is stored in the variable *pos* in the Algorithm 2 and is used in Line 28 and Line 36. This further reduces the time it takes to find the next transaction as the search starts from *pos* as shown in the Algorithm 2 Line 28. The pseudo-code for the execution of each thread while selecting a transaction is present in Algorithm 2.

3.2 Secure Validator

DAG sharing and smart multi-threaded validation have been explored in [6] by Anjana et al. Two important computation errors discussed in [6] are *False Block Rejection* (FBR), where a valid block is incorrectly rejected by a validator, and *Edge Missing BG* (EMB), where an edge is removed from DAG before sharing by a malicious block producer. The solution proposed for the issue of EMB by Anjana et al. in [6] focuses on identifying missing edges between transactions only when they are executed concurrently. However, when a validator executes transactions sequentially, the block may still be accepted. Consequently, a parallel validator would reject the block, while a serial validator would accept it as depicted in the Figure 5. This discrepancy can potentially result in inconsistencies in the final states of the blockchain across different nodes, which is undesirable. To address this issue without sacrificing concurrent block execution, we propose a solution.

A malicious block producer can add extra edges to slow the validator by forcing it to serially execute the block transaction. This case of malicious behaviour is not considered by Anjana et al. [6]. We have denoted the condition as *Extra edge BG* (EEB). In this work, we propose a solution overcoming the drawbacks of the previous solution in resolving FBR and EMB while addressing EEB error.

DAG Validation: The DAG created by the block producer in the blockchain network is shared with the validators in the network. This helps validators save on the time taken for DAG creation. In order to address the issues caused by FBR, EMB, and EEB errors due to DAG sharing, we have proposed *secure validator* for verifying DAGs which is described in Algorithm 3. The secure validator checks for missing edges and extra edges present in the DAG shared. This is performed by multiple threads for swift graph verification. For all the

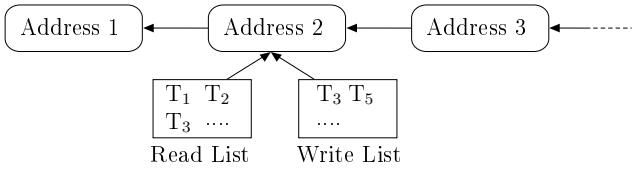

Figure. 6: Linked list address data for secure validator.

Algorithm 3 Multi-threaded secureValidator(): m threads concurrently verify the DAG shared

```
46: procedure SECUREVALIDATOR(DAG)
47:     while !mBlockProducer do
48:         Adds ← addsCounter.get&inc()                    ▷ Claim the next address for analyzing
49:         if Adds > adds_count then
50:             addsCounter.dec()
51:             return                                      ▷ Return if all the address are processed
52:         end if
53:         for i = 0 To lenght(Adds.readList) do           ▷ procedure for checking for missing edges
54:             T_i ← Adds.readList[i]
55:             for j = 0 To lenght(Adds.writeList) do                ▷ read-write dependencies
56:                 T_j ← Adds.writeList[j]
57:                 if !checkEdge(T_i, T_j) then
58:                     mBlockProducer ← True
59:                     return
60:                 end if
61:                 incDeg(T_i, T_j)          ▷ Increment the indegree of lower txn and mark the edge
62:             end for
63:         end for
64:         for i = 0 To lenght(Adds.writeList) do
65:             T_i ← Adds.writeList[i]
66:             for j = 0 To lenght(Adds.writeList) do            ▷ write-write dependencies
67:                 T_j ← Adds.writeList[j]
68:                 if !checkEdge(T_i, T_j) then
69:                     mBlockProducer ← True
70:                     return
71:                 end if
72:                 incDeg(T_i, T_j)          ▷ Increment the indegree of lower txn and mark the edge
73:             end for
74:         end for
75:         for i = 0 to txn_count do                       ▷ procedure for checking for extra edges
76:             if T_i.inDeg ≠ T_i.calDeg then  ▷ if shared indegree is equal to calculated indegree
77:                 mBlockProducer ← True
78:                 return
79:             end if
80:         end for
81:     end while
82: end procedure
```

addresses accessed in the block, a read list and a write list are maintained as shown in the Figure 6. By parsing the transactions in the block, transaction IDs are added to the read and write lists of respective addresses. First, check for missing edges is performed by making sure that transactions in the write list of an address have an edge with all transactions present in the respective read and write lists as shown in Algorithm 3 Line 57. A failed check indicates that the DAG shared has a missing edge. During the check, the number of outgoing edges is calculated for each transaction as in Line 72 Algorithm 3.

From Line 75 to Line 80, we compare the sum of the outgoing edges obtained in the above operation with the in-degree array shared along the block. This function identifies if any extra edges are present in the DAG. As a result, the secure validator verifies the DAG and recognizes malicious block producers (if any). The procedure to handle such nodes depends on the type and functionalities of blockchain technology. This way, we eliminate the FBR, EMB, and EEB errors and validate the DAG shared. Detailed algorithms with extensive explanations can be obtained by referring to appendix C in [16].

4 Experiments Analysis

4.1 Implementation Details

We have chosen Hyperledger Sawtooth as our testing platform since it has good support for parallelism and already has an inbuilt parallel scheduler. To incorporate the DAG framework into the Sawtooth architecture, we have to modify the current parallel scheduler module. Due to the modular nature of Sawtooth, any modifications made to a module can be restricted within the module itself without impacting the remaining modules of the architecture. Ensuring this however requires that the modifications to modules are performed with great care.

We have now implemented the DAG sharing and secure validator modules in Sawtooth 1.2.6. Our modules are in CPP language while the Sawtooth core was developed in both Rust and Python. We have chosen CPP for its efficient support for concurrent programming. For DAG sharing, we have modified the block after the block producer has verified that all the transactions in the block are valid. In Sawtooth 1.2.6 we used the input and output addresses present in the transaction structure. Every transaction in the DAG is represented by a graph node and the outgoing edges indicate dependent transactions. In order to ensure efficient validation, the DAG is also stored in the block [5, 6, 10] and shared across the blockchain network. We have used the dependencies list component of the transaction structure (in Sawtooth) to incorporate DAG into the block.

Initially, we implemented the DAG using a linked list data structure. This is ideal when the size of the graph is unknown and the graph needs to be dynamic. Given that the number of transactions in a block does not change and the limit to the number of transactions a block can contain, we have designed an adjacency matrix implementation for DAG. The results have shown further improvement over the linked list implementation. This is because the adjacency matrix is direct access whereas the linked list implementation would require traversal across the list to reach the desired node.

In Sawtooth 1.2.6 block validators, the secure validator is implemented. The DAG is recreated using the dependencies list provided in the transaction by the block producer. This saves the time taken to create the DAG for concurrent execution again in the validators. The secure validator performs various checks for missing edges that should have been present in the DAG shared by the block producer as explained in SubSection 3.2.

Transaction Families: We implemented four transaction families to test the performance of our approach: (a) SimpleWallet, (b) Intkey, (c) Voting and (d) Insurance. In SimpleWallet one can create accounts, deposit, withdraw and transfer money. In Intkey clients can increment and decrement values stored in different variables. In Voting, the operations are 'create parties', 'add voters' and the voters can 'vote' one of the parties. The insurance family is a data storage transaction family where user details like ID, name, and address details are stored and manipulated.[2] To control the percentage of conflicts between transactions, one must have control over the keys created. We have modified the batch creation technique in these transaction families to allow the user to submit multiple transactions in a batch. This way we can not only just control the number of transactions in a batch but also the conflicts among the transactions in a batch. We individually observed each transaction family behaviour under various experiments and a mix of all four types of transactions in a block.

4.2 Experiments

We have conducted several experiments to extensively test our proposed framework. In order to assess the framework's performance across different scenarios, we have devised three conflict parameters (CP) that indicate the level of dependency among the transaction. The conflict parameters, CP1, CP2, and CP3, are metrics used to assess different aspects of a DAG representing transactions. CP1, measures the proportion of transactions in the DAG that have at least one dependency. It indicates how interconnected the transactions are, with higher values suggesting a greater level of dependencies. CP2, represents the ratio of dependencies to the total number of transactions in the DAG. It provides insights into the density of dependencies within the graph. A higher CP2 value indicates a higher density of dependencies among transactions. CP3, quantifies the degree of parallelism in the DAG by calculating the number of disjoint components, which are subgraphs without interconnections. A lower CP3 value suggests a higher level of parallelism, indicating that transactions can be executed independently in separate components. By evaluating these conflict ratios, one can gain a deeper understanding of the interdependencies and parallelizability of transactions within the DAG.

We have designed four experiments, each varying one parameter while the rest of the parameters are constant. The four parameters are (1) the number of blocks, (2) the number of transactions in the block, (3) the degree of dependency, and (4) the number of threads. The experimental setup is named one to four, respectively. We have named the adjacency matrix implementation of our proposed framework as *Adj_DAG* and linked list implementation as *LL_DAG*. The Sawtooth inbuilt parallel scheduler uses a tree data structure; accordingly, we have named it as *Tree* and serial execution output as *Serial* in our results. We have observed that due to the presence of global lock in Python, the change in the number of

[2] The transaction family code can be accessed here: https://github.com/PDCRL/ConcSawtooth

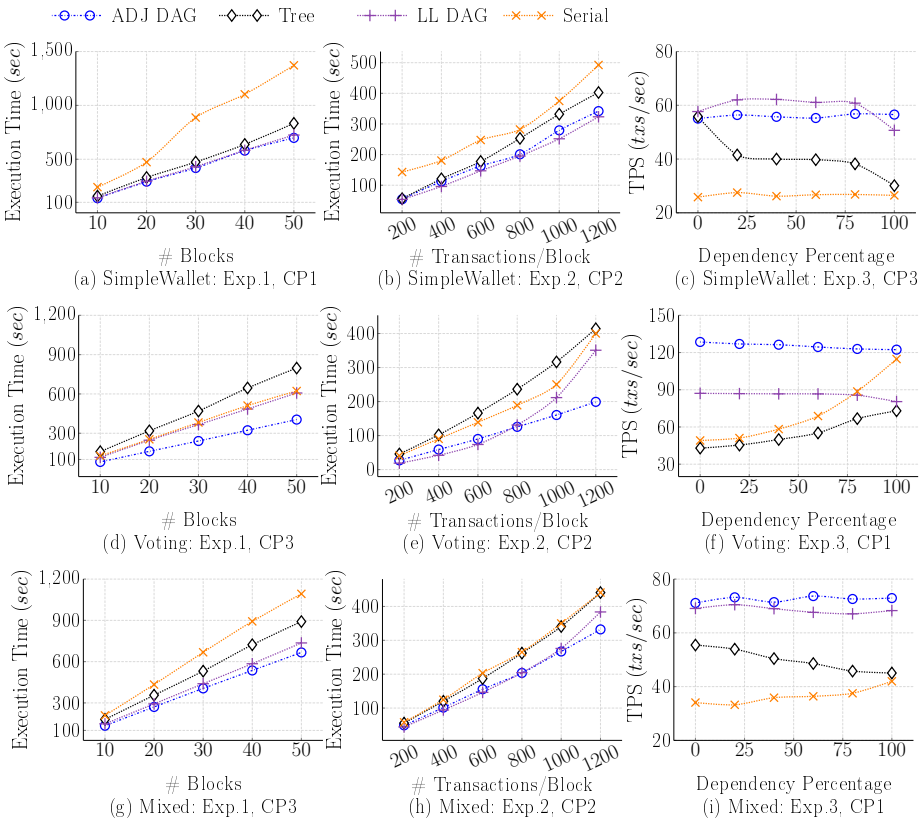

Figure. 7: Detailed analysis of our proposed framework performance with both adjacency matrix and linked list implementations in Sawtooth 1.2.6.

threads has not impacted the performance significantly. Due to this, we have not presented the results of the experiment (4) in this work.

It can be observed from Figure 7 that the adjacency matrix and linked list implementation of our proposed framework perform significantly better than the tree-based parallel scheduling and serial execution. We have illustrated here some of the experiments we have conducted, and the rest can be found in the associated technical report [16] (Appendix D). Figure 7 (a), (d), and (g) illustrate the impact of change in the number of blocks on various schedulers. On average the speedup of *Adj_DAG* over *Serial* is 1.58 times and *LL_DAG* is 1.43 times, while *Tree* is 1.22 times. The average speedup of *Adj_DAG* over *Tree* is 1.29 and *LL_DAG* is 1.17 times.

Experiment (2) results are depicted in Figure 7 (b), (e), and (h). We can observe that the gap between serial and parallel schedulers increases with an increase in the number of transactions in the block. We observe that the higher the number of transactions greater the scope for concurrency. For Experiment (3),

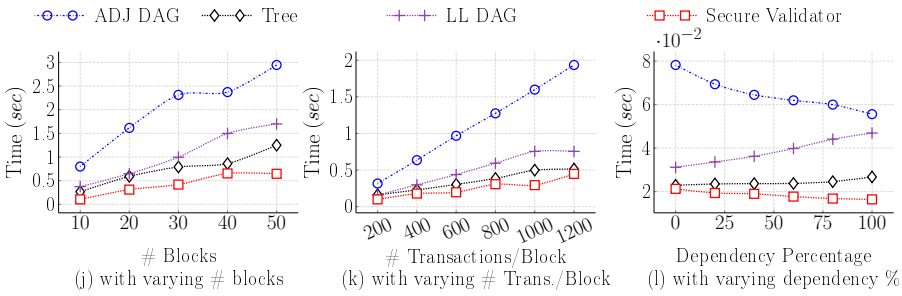

Figure. 8: Comparison of data structure creation time for all the parallel schedulers

we have varied the degree of dependency between the transactions and measured its impact on the transactions per second (TPS). The dependency among the transactions is increased by making multiple transactions access the same accounts/addresses. Ideally, varying the conflict percentage without changing the number of transactions should not impact serial execution throughput. However, a decrease in the number of different memory accesses due to caching improves the execution time. We can observe this phenomenon in serial execution time in Figure 7 (c), (f), and (i). Interestingly these opposing effects, temporal locality, and increase in conflicts balance each other, and a steady TPS is maintained for *ADJ_DAG* and *LL_DAG* algorithms. But, in *Tree* scheduler, the performance further decreases with increased conflicts as it dominates over the temporal locality.

Figure 7 (d), (e), and (f) show the Voting transaction family behavior under experiments (1), (2), and (3). Unlike the other transaction families, *Serial* execution is faster than *Tree* scheduler with this family. We discovered that the reason for this is that the entire list of voters list and parties are accessed for any transaction (operation) in this family instead of the one particular voter and party address. This causes higher overheads which leads to the observation that the design of the transaction family (smart contract) plays a crucial role in performance optimization. One can observe that the *ADJ_DAG* and *LL_DAG* still perform better as they use transactions to represent the dependency data structure, unlike *Tree* scheduler that uses addresses.

The *secure validator* framework efficiently verifies the DAG shared by the block producer and eliminates the need to reconstruct the DAG at every block validator. The execution time of the secure validator and adjacency DAG scheduler will only vary in the dependency graph creation aspect. To highlight the savings achieved through secure validator, we analyzed the dependency data structure creation and verification time for various schedulers in Figure 7. One can observe that the *secure validator* takes the least execution time as seen in the Figure 8 (j), (k), and (l). Figure 8 (l) shows that *secure validator* is stable against the variations in the dependency in the graph. Due to lack of space, the remaining experimental results, including the ones on *Intkey* and *Insurance* transaction families, are described in the technical report [16] (Appendix D).

5 Related Work

In the past few years, blockchain technology has gained tremendous popularity and is used in a wide variety of fields. Although blockchains are capable of offering a variety of advantages, one of the most cited concerns is scalability. Consensus protocols and transaction throughput are the two significant bottlenecks of blockchain performance. In contrast to PoW, alternative consensus protocols like PoS and PoET are introduced to minimize consensus time. However, transaction throughput continues to be a hindrance to scalability. Exercising parallel execution of transactions in a block is one of the solutions to optimize blockchains.

Dickerson et al. [10] introduced the concept of parallel execution of Ethereum [1] transactions using Software Transactional Memory (STM). The block producer executes transactions in the block using STM, and the serializable concurrent state is discovered. This is then shared with the validators to achieve deterministic execution. Following this, there have been multiple STM-based concurrent transaction execution frameworks for blockchains [3, 7, 11]. Besides the significant overhead associated with executing transactions through STMs, transactions sometimes fail due to dependencies and must be re-executed. Another drawback is that they cannot have operations that cannot be undone, which is a significant obstacle to smart contract design. During concurrent execution, STM-based approaches identify conflicts among transactions dynamically, i.e., during execution. This results in various transactions failing or rolling back to resolve the conflict. This has a significant impact on throughput and is not optimal for blocks with high interdependencies. In general, a dynamic approach is ideal, but it is not necessary for blockchains whose addresses are either included in the transactions or are easily inferred. For such systems, we propose a parallel execution framework for transactions in a block.

Sharding is another popular technique to address scaling issues in blockchains. In this, the nodes present in the network are categorized into small groups. Each group processes transactions parallelly with the other groups. Sharding is being explored earnestly as a solution to scalability issues [8, 9, 12, 14, 17, 19, 20]. The criteria for sharding are different in each approach. Few are specifically designed for monetary transactions in blockchains [12, 19]. This leads to smart contract transactions being processed on a single shard leading to an inefficient distribution of computational work. The implementation of transactions that span across smart contracts becomes intricate with sharding. Protocols have to be designed specifically for inter-shard communication, further increasing the complexity of the design [9].

The Aeolus blockchain [20], is specifically tailored for Internet of Things (IoT) devices that face limitations in executing multiple transactions rapidly. Aeolus addresses this challenge by harnessing the computing resources available in a cluster of nodes to reduce the time required for transaction execution, thereby enhancing the overall performance of the blockchain. The sharding technique limits the degree of parallelization to the number of shards irrespective of actual capacity. If the shards are dynamic, in the worst case, the number of shards is

equal to the number of transactions. Sharding is considered unsuitable for transactions with high inter-dependencies. In contrast, we designed an efficient parallel scheduler for blockchain nodes to execute block transactions concurrently. Our proposed approach can be implemented on top of the sharding approach to improve the efficiency of individual nodes within each shard, where transactions in a block can be executed in parallel.

6 Conclusion and Future Work

In this paper, we proposed a concurrent transaction execution framework for blockchains. We proposed a parallel scheduler and a secure validator module for the blockchain node architecture. The parallel scheduler is responsible for identifying the dependencies among the transactions in the block and scheduling them for conflict-free execution. The dependencies are represented by a DAG and are shared along with the block to minimize the validation time of validating nodes. DAGs are validated using the secure validator, which determines if a malicious block producer has shared inaccurate graphs. The proposed approach has been thoroughly tested in Hyperledger Sawtooth 1.2.6 [2] and is flexible enough to be implemented in any blockchain that follows the order-execute paradigm [4]. One possible future step would be to extend the implementation of the proposed approach to different blockchain platforms and compare their performance. Further, fault tolerance and scalability for each blockchain node on its own (i.e., horizontal scaling of each validator node) can be explored.

Acknowledgements: We would like to express our sincere gratitude to the paper and artifact reviewers who dedicated their time and expertise to evaluate our work. We would also like to extend our gratitude to the members of the MeitY and NBF (National Blockchain Framework) project for their support throughout the research.

References

1. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. `https://ethereum.org/`
2. Hyperledger Sawtooth. `https://sawtooth.hyperledger.org/`
3. Amiri, M.J., Agrawal, D., El Abbadi, A.: ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). pp. 1337–1347 (2019)
4. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S.W., Yellick, J.: Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. EuroSys (2018)
5. Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In: PDP. pp. 83–92 (2019)

6. Anjana, P.S., Attiya, H., Kumari, S., Peri, S., Somani, A.: Efficient Concurrent Execution of Smart Contracts in Blockchains Using Object-Based Transactional Memory. In: Networked Systems - 8th International Conference, NETYS. vol. 12129, pp. 77–93. Springer (2020)
7. Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: OptSmart: a space efficient Optimistic concurrent execution of Smart contracts. Distributed and Parallel Databases (May 2022)
8. Baheti, S., Anjana, P.S., Peri, S., Simmhan, Y.: DiPETrans: A framework for Distributed Parallel Execution of Transactions of Blocks in Blockchains. Concurrency and Computation: Practice and Experience **34**(10), e6804 (2022)
9. Dang, H., Dinh, T.T.A., Loghin, D., Chang, E.C., Lin, Q., Ooi, B.C.: Towards Scaling Blockchain Systems via Sharding. In: Proceedings of the 2019 International Conference on Management of Data. p. 123–140. SIGMOD '19, Association for Computing Machinery, New York, NY, USA (2019)
10. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding Concurrency to Smart Contracts. p. 303–312. PODC '17, Association for Computing Machinery, New York, NY, USA (2017)
11. Gelashvili, R., Spiegelman, A., Xiang, Z., Danezis, G., Li, Z., Malkhi, D., Xia, Y., Zhou, R.: Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing (2022)
12. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: Omniledger: A secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 583–598 (2018)
13. Kunz, T., Black, J.P., Taylor, D.J., Basten, T.: POET: Target-System Independent Visualizations of Complex Distributed-Applications Executions. The Computer Journal **40**(8) (1997)
14. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. p. 17–30. CCS '16, Association for Computing Machinery, New York, NY, USA (2016)
15. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf` (2008)
16. Piduguralla, M., Chakraborty, S., Anjana, P.S., Peri, S.: An Efficient Framework for Execution of Smart Contracts in Hyperledger Sawtooth (2023)
17. Valtchanov, A., Helbling, L., Mekiker, B., Wittie, M.P.: Parallel Block Execution in SoCC Blockchains through Optimistic Concurrency Control. In: 2021 IEEE Globecom Workshops (GC Wkshps). pp. 1–6 (2021)
18. Vasin, P.: Blackcoin's proof-of-stake protocol v2. URL: https://blackcoin.co/blackcoin-pos-protocol-v2-whitepaper. pdf **71** (2014)
19. Zamani, M., Movahedi, M., Raykova, M.: RapidChain: Scaling Blockchain via Full Sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 931–948. CCS '18, Association for Computing Machinery, New York, NY, USA (2018)
20. Zheng, P., Xu, Q., Luo, X., Zheng, Z., Zheng, W., Chen, X., Zhou, Z., Yan, Y., Zhang, H.: Aeolus: Distributed Execution of Permissioned Blockchain Transactions via State Sharding. IEEE Transactions on Industrial Informatics **18**(12), 9227–9238 (2022)