DiPETrans: A Framework for Distributed Parallel Execution of Transactions of Blocks in Blockchain*

Shrey Baheti†, Parwat Singh Anjana‡, Sathya Peri‡, and Yogesh Simmhan†
†Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India
‡Department of Computer Science and Engineering, Indian Institute of Technology, Hyderabad, India
†{shreybaheti, simmhan}@iisc.ac.in ‡{cs17resch11004, sathya_p}@iith.ac.in

***Abstract*—In most of the modern day blockchain, transactions are executed serially by both miners and validators; also, PoW is determined serially. The serial execution limits the system throughput and increases transaction acceptance latency, even unable to exploit the modern multi-core resources efficiently.**

In this work, we try to increase the throughput by introducing parallel execution of the transactions using a static analysis based technique. We propose a framework *DiPETrans* for the distributed parallel execution of block transactions. In *DiPETrans*, trusted peers in the blockchain network form a community to execute the transactions and to find the PoW parallelly. The community follows a master-slave approach for parallel execution. The core idea is that master analyzes the transactions using a static analysis based technique, creates different groups (shards) of non-conflicting transactions, and distribute shards to workers (community members) to execute them parallelly. After transaction execution, communities compute power is utilized to find PoW parallelly. On successful creation of a block, the master broadcasts the proposed block to other peers in the network for validation. On receiving a block, validators re-executes the block transactions, either parallelly (community) or serially (solo validators). If they reach the same state as shared by the miner, then accept the block otherwise reject. We proposed two different approaches for the validator. In the first *Sharing Validator*, the miner shares the dependency information (shard) in the block to help validators to execute the transaction parallelly, while in another approach (*Default Validator*) no dependency information is shared with validators and validators need to determine the dependencies using static analysis of the block transactions.

We report experiments using 5,170,597 transactions from the Ethereum blockchain and execute them using our DiPETrans framework to empirically validate the benefits of our techniques over traditional sequential execution. We achieved a maximum speedup of $2.18\times$ for the miner, $2.02\times$ for the validator, without information sharing (i.e., *Default Validator*), and $2.04\times$ for with information sharing validator (i.e., *Sharing Validator*) for 100 to 500 transactions per block, when using 6 workers (including master) in the community.

***Index Terms*—Blockchain, Smart Contracts, Mining Pools, Parallel Execution, Sharding**

I. Introduction

According to a report published by World Economic Forum (WEF) based on a survey on the future of the blockchain technology predicted that $\approx 10\%$ of the Global Gross Domestic Product (GDP) would be stored on the blockchain by 2025 [1]. Furthermore, with an annual growth rate of 26.2%, the capital market of \$2.5 billion in 2016 may touch the \$19.9 billion by 2025 [2], [3]. Many well-known information technology vendors, governments across the world, Internet giants, and banks are investing in blockchain to accelerate research to make distributed ledger technology versatile [2].

A blockchain is a distributed decentralized database which is a secure, tamper-proof, publicly accessible collection of the records organized as a chain of the blocks [4], [5], [6], [7], [8]. It maintains a distributed global state of the transactions in the absence of a trusted central authority. Due to its usefulness, it has gained wide interest both in industry and academia.

A blockchain consists of *nodes* or *peers* maintained in a peer-to-peer manner. The nodes of the network are known as a *miner* or *validator*. *Miner* when proposing a block to be added to the blockchain, while known as *validator* when validating a block proposed by the miner. A *block* consists of a set of transactions, timestamp, block id, nonce, coin base address (miner address), previous block hash, its hash, and other relevant information. Essentially, miner proposes a block and rest all peers of the network validate that block. Later based on majority consensus block is added to the blockchain. Normally, the entire copy of the blockchain is stored on all the nodes of the system. *Clients* (also known as users) external to the system use the services of the blockchain by sending requests to the nodes of the blockchain systems.

Bitcoin [4], the first blockchain system proposed (by Satoshi Nakamoto), is the most popular blockchain system till date. It is a cryptocurrency system which is highly secure where the users need not trust anyone. Further, there is no central controlling agency like current day banking system. Ethereum [7] is another popular blockchain currently in use and provides various other services apart from cryptocurrencies. As compared to Bitcoin, Ethereum provides support for user-defined programs (scripts), called *smart contracts* [9] to offer complex services.

Smart contracts in the Ethereum are written in Turing complete language Solidity [9]. These contracts automatically provide several complex services using pre-recorded terms and conditions in the contract without the intervention of a trusted third party. In general, a smart contract is like an object in object-oriented programming languages, which consists of methods and data (state) [5], [9]. A client request, aka transaction, consists of these methods and corresponding input parameters. A transaction can be initiated by a client (user) or a peer and executed by all the peers.

A client wishing to use the services of the blockchain, contacts a node of the system and send the request, which

*Authors equally contributed to this work.

is a transaction. A node N_i on receiving several such transactions from different clients or other peers, forms a *block* to be added to the blockchain. The node N_i is said to be a *block-producer* and cryptocurrencies based blockchains such as Bitcoin, Ethereum, block-producers are called as *miners* as they can 'mine' new coins.

Drawback with Existing System: The drawback with the existing blockchain system is that miners and validators execute the transactions serially and so is computing the PoW. Further, finding the PoW [10] is highly computationally intensive random process which requires a lot of computation to create a new block. PoW is required to delay-wait the network to reach consensus and allow miners to create a new block in their favor. But the problem is that the high computation requirements make it very difficult for a resource constraint miner to compete for the block creation to get intensive [11].

Also, in the current era of distributed and multi-core system, sequential transaction execution results in poor throughput. Dickerson et al. [5] observed that the transactions are executed serially in two different contexts. First, they are executed serially by the miner while creating a block. Later, validator re-executes the transactions serially to validate the block. Serial execution of the transaction leads to poor throughput. Also, block transactions are executed several times by the miners and many many times by the validators.

In addition to these problems, due to the substantial high transaction fee, poor throughput (transactions/second), significant transaction acceptance latency, and limited computation capacities prevent widespread adoption [12], [13]. Hence adding parallelism to the blockchain can achieve better efficiency and higher throughput.

Solution Approach: There are few solutions proposed and used in Bitcoin and Ethereum blockchain to mitigate these issues. One such solution is that several resource constraint miners form a pool (community) known as *mining pool* and determines the PoW, and after block acceptance, they share the incentive among them [11], [14], [15], [16], [17], [18], [19]. In the rest of the paper, we use pool and community interchangeably.

Other solutions [5], [20], [21], [22] suggest concurrent execution of the transactions at runtime. These are done in two stages: first, while proposing the block, and second while validating the block. This helps in achieving better speedup in creating the block and its acceptance, and hence increase the chance of a miner to receive their fees. However, it is not straight forward and requires a proper strategy so that a valid block should not be rejected due to false block rejection or *FBR error* [22]. These techniques follow Software Transactional Memory based runtime techniques to execute transactions concurrently. Miner concurrently executes the block transactions and construct the block graph alongside. The block graph is used to record the dependencies between the transactions where vertices depict the transactions and edges between them stores the dependencies. In the end, miner adds a block graph in block to help the validator to execute transaction concurrently and to avoid FBR error. During the concurrent execution at validator, the FBR error may easily occur if transactions dependencies are not recorded appropriately in the block graph.

In this work, our objective is to execute the transactions in parallel based on a static analysis of the transactions, using a community of trusted nodes (workers) modeled as a master and slaves, along with finding the PoW in parallel, to improve the performance of block creation and validation. We proposed a framework for distributed parallel execution in a formal setting inspired by Ethereum [7]. We follow a static analysis based sharding technique to determine the transaction dependencies. Hence, *FBR error* will not occur if the validator does not use information shared by the miner as dependencies remain the same in the statical analysis. A validator can perform the static analysis before parallel execution, which makes it very straightforward to adopt.

Unrelated transactions of the block are grouped into different shards (see Fig. 2), and shards are assigned to different nodes of the community (see Fig. 1), which the nodes execute independently in parallel. To our knowledge, this is the first work which uses static analysis for identifying block transactions that can be executed in parallel, and combines these with benefits associated with sharding and mining pools.

*The **major contributions** of this paper are as follows:*

- We propose a framework *DiPETrans* for parallel execution of the transactions at miners and validators based on transaction sharding. This work is the first contribution which uses static analysis based transaction sharding to execute the transactions of the blocks in mining pools parallelly.
- We implemented two different approaches for the validator. In the first approach known as *Sharing Validator*, the miner incorporates some information about the dependent transactions (shards) in the block to help the validators to execute the block transactions deterministically and in parallel. While in the second *Default Validator*, the miner does not include dependency information in the block, and validators infer the dependencies themselves.
- We report experiments using 5,170,597 transactions from the Ethereum blockchain and execute them using our DiPETrans framework to empirically validate the benefits of our techniques over traditional sequential execution. We achieved a maximum speedup of $2.18\times$ for the miner, $2.02\times$ for the validator, without information sharing (i.e., *Default Validator*), and $2.04\times$ for with information sharing (i.e., *Sharing Validator*) for 100 to 500 transactions per block, when using 6 workers (including master) in the community.

The rest of the paper is organized as follows: We present the background and related work in *Section II*. While in *Section III*, we describe the proposed *DiPETrans* architecture and methodology *Section IV* consist of the experimental evaluation and results. Finally, we conclude with some future research directions in *Section V*.

II. Background and Related Work

This section presents the background and overview of existing techniques for parallel execution of the transactions. We first introduce the working of blockchain technology and then summaries the work on concurrent execution of the smart contract transactions. Then, we present the work on mining pools and sharding techniques closest in spirit to this work.

Background: In most of the popular blockchain systems such as Bitcoin and Ethereum, transactions in a block are executed in an ordered manner first by the miners later by the validators [5]. When a miner creates a block, the miners typically choose transactions to form a block from a pool based on their preference, e.g., giving higher priority to the transactions with higher fees. After selecting the transactions the miner (1) serially executes the transactions, (2) adds the final state of the system to the block after execution, (3) next find the proof-of-work (PoW) [10] and (4) broadcast the block in the network to other peers for validation to earn the reward. PoW is an answer to a mathematical puzzle in which miner tries to find the hash of the block smaller than the given difficulty. This technique is used in the Bitcoin and Ethereum.

Later after receiving a block, a node validates the contents of the block. Such a node is called the *validator*. Thus when a node N_i is block-producer, every other node in the system acts as a validator. Similarly, when another node N_j is the miner, then N_i acts as a validator. The validators (1) re-execute the transactions in the block received serially, (2) verify to see if the final state computed by them is same as the final state provided by the miner in the block, and (3) also validates if the miner solved the puzzle (PoW) correctly. The transaction re-execution by the validators is done serially in the same order as proposed by the miner to attain the consensus [5]. After validation, if the block is accepted by the majority (accepted by more than 50%) of the validators, then the block is added to the blockchain, and the miner gets the incentive (in case Bitcoin and Ethereum).

Further, blockchain is designed such a way that it forces a chronological order between the blocks. Every block which is added to the chain depends on the cryptographic hash of the previous block. This ordering based on cryptographic hash makes it exponentially challenging to make any change to the previous blocks. In order to make any small change to already accepted transactions or a block stored in the blockchain requires recalculation of PoW of all the subsequent blocks and acceptance by the majority of the peers in the network. Also, if two blocks are proposed at the same time and added to the chain, they form a *fork*. To resolve the forks, the branch with the longest chain is considered as the final. This allows mining to be secure and maintain a global consensus based on processing power.

Related Work: Since the launch of Bitcoin in 2008 by Satoshi Nakamoto [4] the blockchain technology has received immense attention from research communities both from academia and industries. Blockchain is introduced as a digital distributed decentralized system of cryptocurrency currency. However, decentralized digital currencies have been introduced long back before Bitcoin in eCash [23] and later, the peer-to-peer currency in [24], [25]. But none of them concentrated on having a common global log stored at every peer. Bitcoin introduces this concept as distributed decentralized highly secure ledger as blockchain technology. Later Ethereum [7] started using smart contracts (a user-defined computer program) in the blockchain. Which further enhanced the potential of blockchain technology from being a ledger technology to more generalized technology for *anything of values*. Due to the use of smart contracts in the blockchain, it becomes versatile and adopted for many applications such as digital identity, energy market, supply-chain, healthcare, real estate, asset tokenization, etc. Following that many blockchain platforms have been developed based on smart contracts such as EOS [12], Hyperledger [8], MultiChain [26], OmniLedger [27], and RapidChain [28].

However, due to the substantially high transaction fee, poor throughput, high latency, and limited computation capacities prevent widespread adoption of the public blockchain [12]. Also, most of the existing blockchain platform executes the transaction serially one after another and fails to exploit the current day multi-core resources [5], [20]. Therefore to improve the throughput and to utilize concurrency available with multi-core systems, researchers developed the solutions to execute the transaction parallelly.

For the concurrent execution of the smart contract, Dickerson et al. [5] and Anjana et al. [20], [22] proposed Software Transactional Memory based multi-threaded approaches. They achieved better speedup over serial execution of the transactions; however, their techniques are also based on the assumption of non-nested transaction calls. Saraph et al. [6] performed an empirical analysis and exploited simple speculative concurrency in Ethereum smart contracts, for this, they grouped the transactions of a block into two groups one consist of non-conflicting transactions that can be executed parallelly while other consists of conflicting transactions and executed serially. They proposed a lock based technique to avoid the inconsistency that may occur due to concurrent execution. In another contribution, Zang et al. [21] proposed a Multi-Version Timestamp Ordering Concurrency Control based concurrent validator. In their approach, the miner can use any concurrency control protocol and generates the read-write set to help the validators to execute the contract transaction of a block concurrently. In [29], Bartoletti et al. presented a statical analysis based theoretical perspective of concurrent execution of the smart contract transactions.

Distributed mining pools in Bitcoin and Ethereum network make use of distributed compute power to find the PoW parallelly and share the incentive based on the pre-agreed mechanism (proportional, pay-per-share or pay-per-last-N-shares, etc.) [15], [16], [17]. The distributed mining pool based centralized and decentralized solutions are practically implemented and utilized for determining PoW in both Bitcoin and Ethereum. In Bitcoin network, approx 95 percent mining power resides with less than 10 mining pools while in

Ethereum network, roughly 80 percent mining power held by 6 mining pools [30]. In distributed mining pools, computation constraint miner participates in the mining to earn the rewards which they can't achieve independently.

Furthermore, in sharding [31] based techniques either overall system is partitioned into smaller equally sized committees or data (blockchain) is partitioned in such a way that a new node need not to validate entire chain instead validate only specific block from the chain. In EOS blockchain [12], concept of sharding is proposed to be implemented for parallel execution of the transaction. The static analysis of the block can be performed to determine the non-conflicting transactions in a block [29], [31], [32]. The two transactions that do not modify the common data item (account) grouped into different shards and different shards of the block executed parallelly. So transactions of a block belong to different shards executed concurrently using multiple threads first by the miner, later during validation by the validators. EOS [12], OmniLedger [27], and RapidChain [28] proposed to provided optimization using sharding technique.

In this work, the idea of the mining pools is taken up to parallelize the transactions execution, mining, and validation. Essentially, we proposed to use these two concepts, static analysis based *transaction sharding*, and *distributed mining pool* to execute the transactions parallelly along with parallel PoW computation.

III. DiPETrans Architecture

This section presents the proposed *DiPETrans* framework. We first introduce a high-level overview of the architecture that gives the functionalities of the miner and the validator. Following that, the master-slave approach of a mining community is illustrated. Finally, the algorithms for static analysis of transactions and distributed mining are explained.

A. DiPETrans Architecture

The architecture of the *DiPETrans* framework is shown in *Fig. 1*. There are different mining communities, such as *Community 1 to 4* (Fig. 1, ②). Each community is a set of devices (workers) which use their distributed compute power collaboratively to execute transactions and solve the PoW in parallel for a block. Workers in a community trust each other. As with existing mining pools, all workers in a community that participates in the parallel mining of a block get a part of the incentive fee, based on pre-agreed conditions.

One of the workers in the community is identified as the *Master* while the others are *Slaves*. The master serves as the peer that represents the community with the blockchain network for all operations. When a user submits their request (Fig. 1, ①) to one of the peers in the blockchain network, the transaction will be broadcast to all peers in the network, including the master of each community, and be placed in their pending transaction queue (Fig. 1, ③). Then all the miners in the network compete to form the next block from these transactions.

1) Functions of the Miner: We followed a master-slave model within the community, and the master node is responsible for coordinating the overall functionality of the community (Fig. 1 ④). The master can be selected based on a leader election algorithm or on some other approach. We assumed that no workers fail within the community. When the community acts as a miner to create new blocks, there are two phases: one is transaction execution (ⓘ), and the second is solving the PoW (ⓘⓘ). Both of these are parallelized. When the community acts as a validator, it only executes the first phase of executing transactions to validate them. In the miner's first phase, the master selects the transactions from the pending transaction queue of the community (③) to construct a block (⑤). Then, it identifies the independent transactions by performing a static analysis of the transactions (discussed later in Alg. 1). It groups dependents transactions into a single shard, and independent ones across different shards (⑥). The master then sends the shards to the slaves, along with the current state of the accounts (stateful variables) accessed by those transactions (⑦, ⑧).

On receiving a shard from the master, the slave worker executes the transactions present in its shard(s) serially (⑦), computes the new state for the accounts locally, and sends the results back to the master. While transactions across shards are independent, those within a shard will have dependencies (Fig. 2), and hence are executed sequentially. To improve the throughput further, one can perform concurrent execution of transactions within a shard based on Software Transactional Memory (STM) to leverage multi-processing on a single device [5], [20], [22]. This is left as a future extension. Once all slaves complete execution of the shards assigned to them, the master computes the final state of the block from the local states returned by the slaves.

In the PoW phase (⑨, ⓘⓘ) of block creation, the master sends the block (header along with transactions), and different nonce ranges to the slaves to find the block hash that is smaller than the required difficulty. This is a brute-force iteration over different possible values of the nonce unit a match is found, and forms the PoW. Different slaves operate on different ranges of values in parallel to find the block hash. When a slave finds the correct hash, it informs the master. The master node then notifies the remaining workers to terminate their computation. The master proposes the block with the executed transactions and the PoW, updates its local chain (⑩), and broadcasts it to all peers in the blockchain network for validation. A successful validation by a majority of peers and addition of the block to the consensus blockchain will result in the workers of the mining community receiving the incentive fee for that block based on a pre-agreed condition.

2) Functions of the Validator: After receiving a block from a miner (Fig. 1, Ⓐ), the remaining peers of the network serve as its validators. They validate the block by re-executing the transactions present in the block and check if the PoW hash matches. Verifying the PoW hash is not very expensive. When a DiPETrans community acts as a validator, the devices follow a similar approach as the first phase of mining (Fig. 1 Ⓐ –

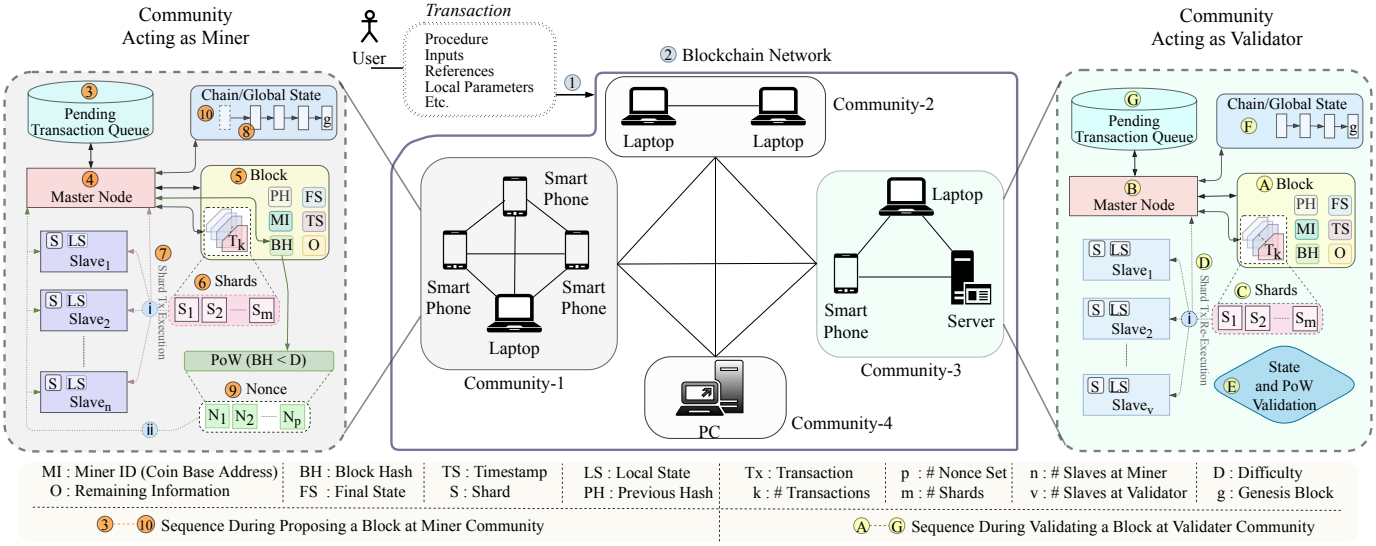

Fig. 1: Overview of the DiPETrans Architecture and Functions

Ⓖ, ⓘ). However, the only difference is that validators do not find the PoW ⓘⓘ, instead verify whether miner has done sufficient work to finding the correct PoW, and validate the final state computed by them based on their local chain with the final state supplied by the miner in the block (Fig. 1 Ⓔ). Alternatively, a validator can also execute the transactions serially if they are not part of any community.

We take two different approaches for the validation. In the first, miner offer hints on the dependency information as part of the transactions in the block with the validators. Specifically, the miner includes the shard ID for each transaction in the *other* field of the block (Fig. 1 Ⓞ), and this can directly be used by the validator to shard the transactions for parallel validation. This avoids a call to Alg. 1 by each validator in the blockchain network. We refer to these as *Sharing Validators*. The second approach is the *Default Validator*, and here no additional details about shards are included in the block. The validator, if part of a community, may use the same Alg. 1 for static analysis of the dependencies themselves, or if a stand-alone validator may validate the transactions sequentially.

3) Sharding of the Block Transactions: Sharding is the process of identifying and grouping the dependent transactions in a block, with one shard created per group. This is illustrated in Fig. 2. Transaction T_1 accesses the account (stateful variables) A_1 and A_3, T_5 accesses A_1 and A_8, while T_7 accesses A_2 and A_3. Since T_1, T_5, and T_7 are accessing common accounts, they are dependent oneach other and grouped into the same shard, $Shard_1$. Similarly transactions T_2, T_3, and T_9 are grouped into $Shard_2$, while T_4, T_6, and T_8 are grouped into $Shard_3$. Transactions in each shard are independent from those in other shards, and each shard can be executed in parallel by different slaves of the community.

We model the problem of finding the shards as a graph problem. Specifically, each *account* serves as a *vertex* in the *transaction dependency graph*, identified by its *address*, and

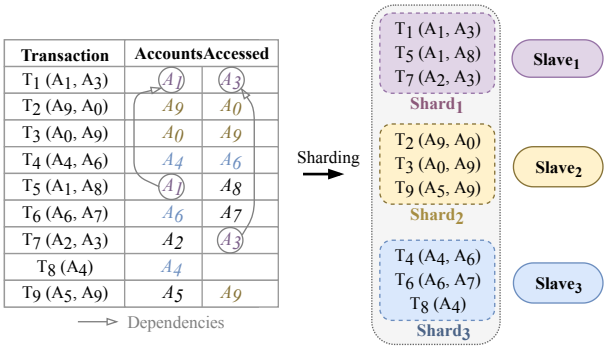

Fig. 2: Sharding of the transactions in a block

Algorithm 1: Analyze()

```
Data: txnsList
Result: sendTxnsMap
Procedure Analyze(txnsList):
    // prepare AdjacencyMap, ConflictMap,
       AddressList to find WCC
    Map<address,List<txID>> conflictMap;
    Set<address> addressSet;
    Map<address,address> adjacencyMap;
    for tx ∈ txnsList do
        conflictMap[tx.from].put(tx.txID);
        conflictMap[tx.to].put(tx.txID);
        addressSet.put(tx.from);
        addressSet.put(tx.to);
        adjacencyMap[tx.from].put(tx.to);
        adjacencyMap[tx.to].put(tx.from);
    Map<shardID, Set<txID>> shardsMap;
    // Call to WCC till all addresses are visited
    shardsMap = WCC (addressSet,conflictMap);
    Map<workerID,List<Transaction>> sendTxnsMap;
    // equally load balance the shards for slaves
    sendTxnsMap = LoadBalance (shardsMap);
```

we introduce an *undirected edge* when a transaction access two accounts, identified by its *transaction ID*. A single transaction accessing n addresses will introduce $\frac{n(n-1)}{2}$ edges. Next,

we find the *Weakly Connected Component (WCC)* in this dependency graph. Each connected component forms a single shard, and contains the edges (transactions) that are part of that component. The transactions within a single shard are present in their sequential order of arrival. Transactions that are not dependent on any other transaction are not present in this graph, and are placed in singleton shards.

The number of shards thus created may exceed the number of slaves. In this case, we attempt to load balance the number of transactions (shards) per device. Here, we assume that all transactions take the same execution time, which may not be true in practice since smart contract function calls may vary in latency, and be costlier than non-smart contract (monetary) transactions.

B. Sequence of Operations

Fig. 3 shows the sequence diagram for processing a block by a miner and a validator community in *DiPETrans*. There are 4 roles as *MasterMiner*, *SlaveMiner*, *MasterValidator*, and *SlaveValidator*. The *MasterMiner* starts the block execution by creating a block from the transaction queue. The created block consists of a set of transactions (b), including block specific information such as timestamp, miner detail (coin base address), nonce, hash of the previous block, final state, etc. The transactions of the block are formed into a dependency graph for static analysis using WCC (c) to identify disjoint sets of transactions that form shards. Load balancing and mapping of shards to slaves is done as well. The *MasterMiner* then sends the shards for each slave to the devices in parallel (d) and these are executed locally on each slave (e). After successful execution, each *SlaveMiner* sends the updated account states back to the *MasterMiner* (f). The *MasterMiner* updates its global account state based on the responses received from all the *SlaveMiner*s (g).

Once all *SlaveMiner* complete executing their assigned, the *MasterMiner* switches to the PoW phase. It assigns them the task to find the PoW for different ranges of nonces concurrently (h). The *SlaveMiner* searches the range to solve the PoW for the block (i) and sends back a response either when the PoW is solved or their nonce range has been completely searched (j).

Finally, the *MasterMiner* broadcasts the block containing the transactions, the updated account states, the PoW, and optionally the mapping from shards to transactions to the peers in the blockchain network for validation (k).

When a *MasterValidator* receives a block to verify, it needs to re-execute the block transactions and match the resulting account states with those present in the block. For this *MasterValidator*, either use the shard information present in the block (shared validator) or if not present, determine it using the same dependency graph approach as the *MasterMiner* (c). Then *MasterValidator* assigns the shards to the *SlaveValidator s* (l). After successful execution of the transactions assigned by *MasterValidator* (m), each *SlaveValidator* returns the account states back to the *MasterValidator* (n). The responses are verified by the *MasterValidator* with the states present in the

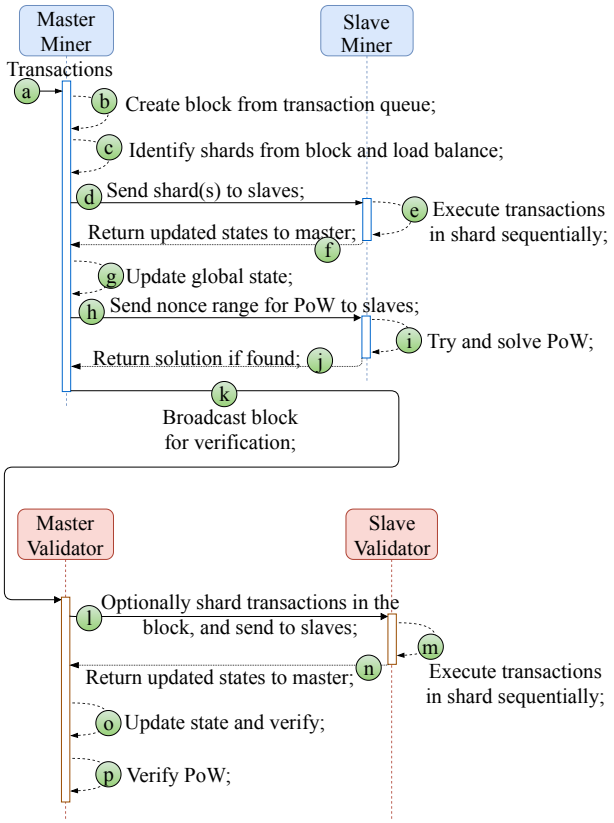

Fig. 3: Sequence diagram of operations during mining and validation

block (o). The *MasterValidator* also confirms that miner has correctly found the PoW (p). After successful verification of both these checks, the *MasterValidator* accepts the block and propagates the message to reach the consensus.

IV. Experiments and Results

In this section we first provide a general overview of *implementation detail*; then we present the *transactions workload and experimental setup*; and in the end, *performance analysis* presents the analysis based on execution time and speedup achieved by the proposed approach over the serial.

A. Implementation

Incorporating our proposed approach into an existing blockchain framework like Ethereum is time-consuming due to the complexity of the codebase of these current platforms. Instead, we implement a stand-alone version of a blockchain framework that models the peers as a set of micro-services that perform the various mining and validation operations that are essential to a blockchain network. This also includes the operations proposed in DiPETrans. The implementation is in *C++* using the Apache thrift cross-platform micro-services library.

B. Transactions Workload

We use historic transactions from the Ethereum blockchain in our experiments. These are acquired from the public-

data archive available on Google's Bigquery Engine [33]. We have chosen the transactions starting from the block number $4,370,000$, which forms a hard fork when Ethereum changed the mining reward from 5 to 3 Ethers. We extract $\approx 80K$ blocks containing $\approx 5M$ transactions. While the original transactions had 17 fields, we selected 6 fields of interest as part of our workload. These include the `from_address` of the sender, the `to_address` of the receiver, `value` transferred in *Wei*, the unit of Ethereum currency, `input` data sent along with the transaction, `receipt_contract_address` which is the contract address when it is created for the first time, and `block_number` where this transaction was present in.

There are two types of transactions we consider: *monetary transactions* and *smart contract transactions* [6]. In the former, also known as value transfer or non-contractual transaction, *coins* are transferred from one account to another account. This is a simple and low-latency operation. In a contractual transaction, one or more smart contract functions are called. As we analyzed the Ethereum transactions, we found out there are 127 unique functions in $20K$ contracts, out of which top 11 (most number of times called) function calls covers $\approx 80\%$ of contract transactions. We re-implement these contract functions from the Solidity language used by Ethereum into C++ function calls that can be invoked by our framework.

Of the $5,170,597$ transactions present in the Ethereum blocks we consider, $193,959$ are contract transactions. However, the wider use of smart contracts will see them have a larger fraction in future, compared to just monetary transactions. Contract transactions are also more compute intensive to execute than the monetary transactions, and benefit more from our parallelism. Hence, we create workloads with different ratios between the contract and monetary transactions: $\{\frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}\}$. Each block formed by our miners have 100 to 500 transactions that are in this ratio, depending on the workload used in an experiment (see Table I Appendix A).

C. Experimental Setup

We used a commodity cluster to run the master and slaves in the mining and validation communities for our DiPETrans blockchain network. Each node in the cluster has an 8-core AMD Opteron 3380 CPU with 32 GB RAM, and are connected using 1 Gbps Ethernet. A mining community has a master running on one node, and between one to five slaves each running on a separate node, depending on the experiment configuration. Similarly, a validation community has one master and between one to five slaves.

D. Performance Analysis

For each simulated data, blocks are executed for serial and 1 to 5 slaves configurations. The serial results serve as the baseline for comparing the performance. Block Execution time is broken down to compare the time taken by transaction execution time along with end-to-end execution time on per block basis. Block execution with mining gives end-to-end block creation time at the miner while without mining provides the transaction execution time at the miner. However, transaction execution time at validator includes the time taken by transaction re-execution and verification. For *Default Validator* time taken by static analysis also included as the part of the time taken by the validator.

We selected 3 different workloads for the analysis and average total time required over all the blocks. In **Workload-1**, the number of transactions varies from 100 to 500, while execution time is averaged across the data set runs (contract : non-contract transactions = 1:1, 1:2, 1:4, 1:8, 1:16). In **Workload-2**, the data set varies from 1:1 to 1:16, and transactions remain fixed to 500 per block. While, in **Workload-3**, transactions is fixed to 500 per block, and community size varies from 1 to 5. For all this analysis, we computed execution time and speedup (with and without mining). The analysis for *Workload-3* and tables for execution time and speedup is given in *Appendix C*.

1) Block Execution Time without Mining: This section presents the experimental analysis done for transaction execution time at miner without mining and at validator. In all the figures, serial execution served as a baseline. The *subfigure (a)* shows the line plots for mean transactions execution time taken by the miner. *subfigure (b)* shows the transactions execution time taken by *Default Validator* while *subfigure (c)* shows transactions execution time taken by *Sharing Validator*. The *subfigure (d)* shows comparison for average transactions execution time taken by *Default Validator* and *Sharing Validator*.

Workload-1: Fig. 4 shows the average transactions execution time taken by the miner and validator (Table III Appendix C) . As shown in the Fig. 4(a), Fig. 4(b), Fig. 4(c), and Fig. 4(d) the time required to execute transactions per block increases as the number of transactions increases in a block. Also, the 1 slave is performing worst due to the overhead of static analysis and communication with the master. Other slave configurations from 2 to 5 are all better than serial execution.

Fig. 4(d) shows the line plot comparison for mean transactions execution time taken by *Default Validator* and *Sharing Validator*. The only difference between these two validators is that *Default Validator* needs to run static analysis on block transactions before execution. The *Default Validator* is supposed to take more time compared to *Sharing Validator*. But the experiment shows no significant difference (can be observed in Table III) as the static analysis is taking very less time.

Workload-2: In this workload, transactions per block are fixed to 500. Fig. 5 (Table IV Appendix C) shows the line plots of mean transactions execution time taken by the miner and validator. In Fig. 5(a), 5(b), and *5(c)*, it can be seen that by varying the ratio of contractual to non contractual transaction i.e., when the number of contract transactions decreases per block, the overall time required to execute transactions also decreases because contractual transaction includes the external calls.

Another observation is that serial execution outperforms 1 slave configuration. Because in 1 slave configuration, there

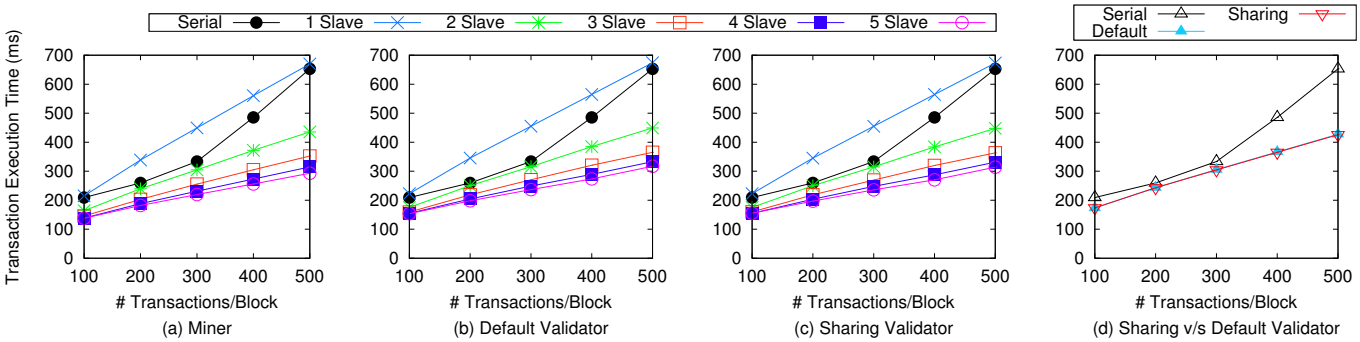

Fig. 4: Workload-1: Average Transaction Execution Time by Miner (Without Mining) and Validator

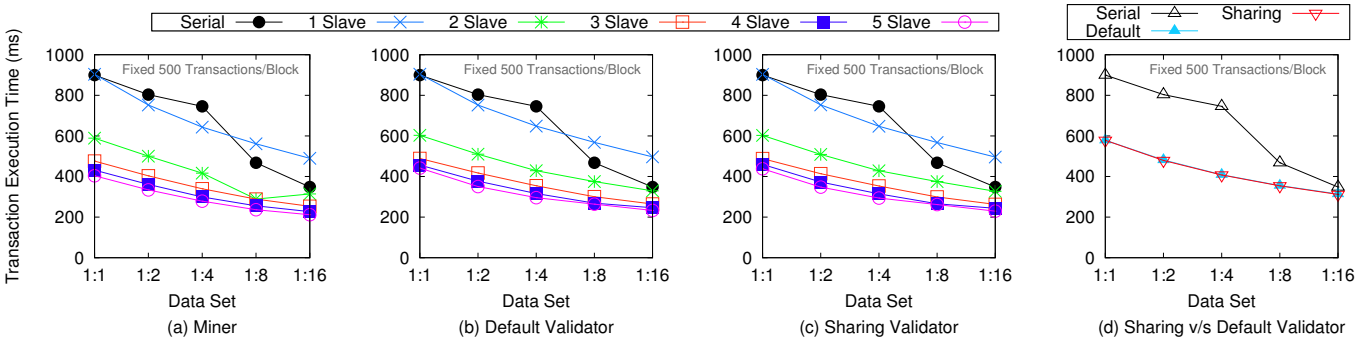

Fig. 5: Workload-2: Average Transaction Execution Time by Miner (Without Mining) and Validator

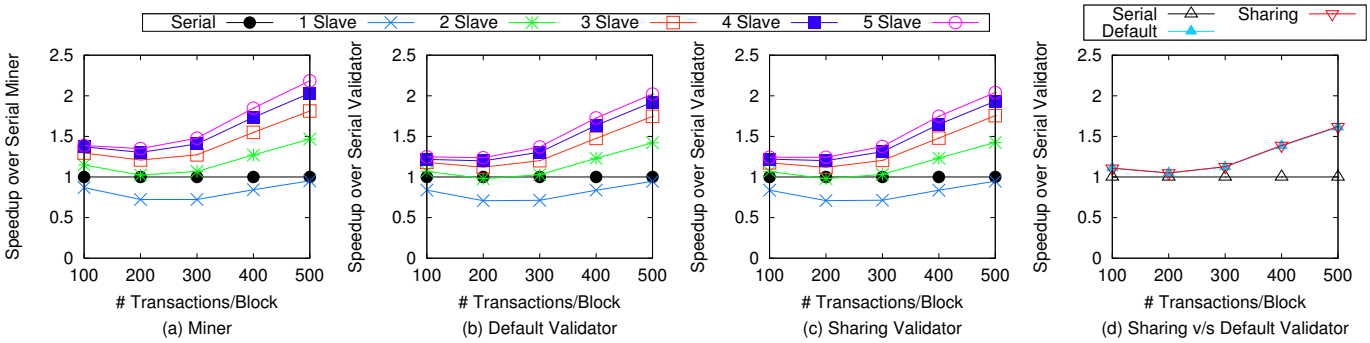

Fig. 6: Workload-1: Average Speedup by Miner (Without Mining) and Validator for Transaction Execution

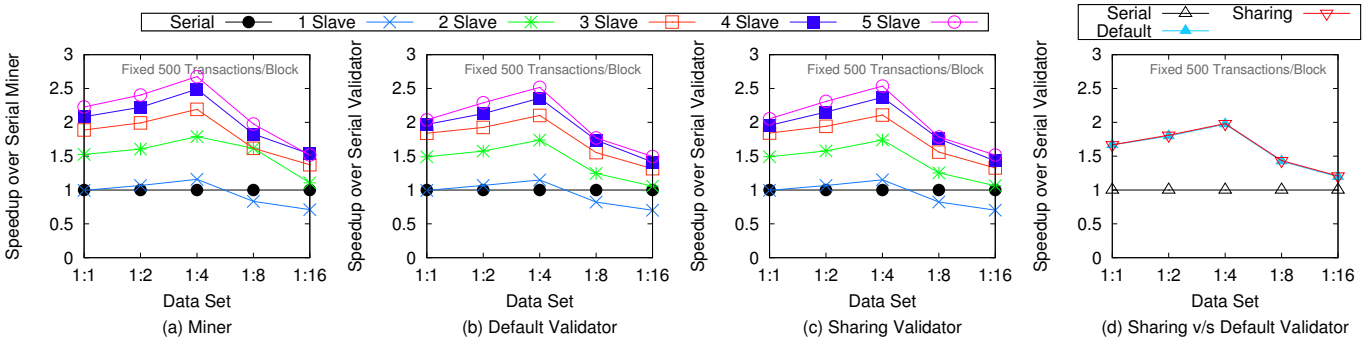

Fig. 7: Workload-2: Average Speedup by Miner (Without Mining) and Validator for Transaction Execution

will be overhead of static analysis and communication with the master. However, all other slave configurations from 2 to 5 performance increases with an increase in the number of slaves in the community. But with the increase of non-contractual transaction in the block serial execution started giving better performance because it may be possible that communication time increases in the proposed parallel approach. In Fig. 5(c), we can see that the time required to execute more transactions

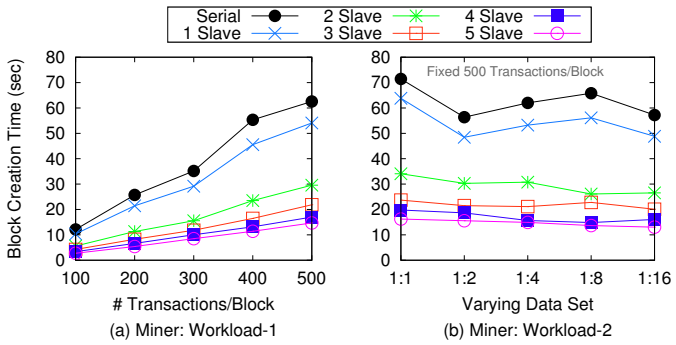

Fig. 8: Average End-to-End Block Creation Time by Miner with Mining

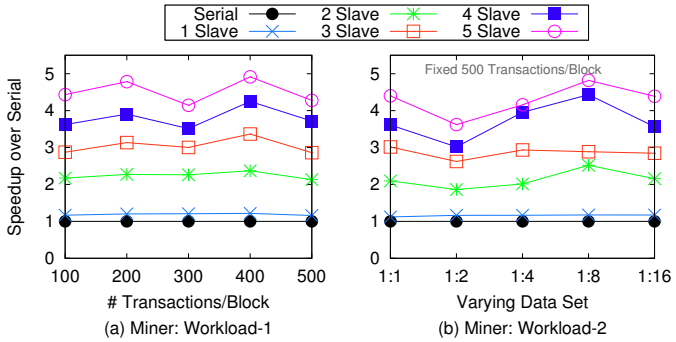

Fig. 9: Average End-to-End Block Creation Speedup by Parallel Miner over Serial Miner with Mining

per block decreases as the number of contract transactions decreases.

Fig. 5(d) shows that the parallel validator is always taking less time than serial, and we can also observe a significant gap when number of non-contractual transactions per block increases. However, the *Default Validator* is supposed to take more time compared to *Sharing Validator*. But the experiment shows that the analyze function is not taking much time, so there is no much difference between them.

Speedup Analysis: The speedup for *Workload-1, Workload-2,* and *Workload-3* is given in the Fig. 6 (Table V), Fig. 7, and Fig. 11 (Table VI) respectively. In all the experiments, serial execution is considered as a baseline, and parallel execution speedup is shown in the form of the line chart. We achieved $\approx 2\times$ speedup, using just 5 slaves in the community. It can be observed that speedup is linearly increasing from 2 slaves to 5 slaves, so it will increase further if the community size increases or the number of transactions per block increases. However, 1 slave community is not achieving any speedup over serial due to static analysis and communication cost.

Further, *Sharing Validator* was assumed to outperform *Default Validator*, but their performance is almost the same, which means static analysis is not taking much time. We have achieved on an average $1.33\times$, $1.255\times$, and $1.259\times$ speedup respectively for parallel miner, *Default Validator* and *Sharing Validator*. Achieved max speedup of $2.18\times$ for miner, $2.02\times$ for *Default Validator* and $2.04\times$ for *Sharing Validator* for 100 to 500 transactions per block. Experiments show that parallel execution gives better speedup and can be used by the trusted node to form a community to receive the incentive by mining the block in the blockchain.

2) Block Execution Time with Mining: Here we present the analysis for end-to-end block creation time at miner with mining. We have observed that with the increase in the number of transactions per block end-to-end time is increased for fining the PoW. This is because the difficulty is fixed for all the experiments at the start of the experiments. Finding PoW is a random process so we cannot guarantee the maximum time to find PoW.

Existing blockchain platforms calibrate the difficulty to keep the mean end-to-end block creation time within limits like in Bitcoin; the block is created every 10 minutes and every two weeks 2016 blocks. After every 2016 blocks, the difficulty is calibrated based on how much time is taken if it has taken more than two weeks, the difficulty is reduced otherwise increased also several other factors are also considered to change the difficulty. While in Ethereum blockchain, roughly every 10 to 19 seconds a block is produced, so the difficulty is fixed accordingly. After every block creation, if mining time goes below 9 seconds, then *Geth* a multipurpose command line tool that runs Ethereum full node tries to increase the difficulty. In case when the block creation difference is more than 20, Geth tries to reduce the mining difficulty of the system. In the proposed *DiPETrans* framework throughout the experiments, we fixed the difficulty, and due to that reason with an increase in transaction per block increase in mining time can be observed.

As shown in Fig. 8 and Fig. 9, when the number of slaves increases in the community end-to-end block creation time decreases. In *Workload-1* with an increase in the number of transactions per block time taken by mining algorithm also increases, as shown in 8(a). While as shown in Fig. 8(b) when monetary transaction increases per block mining time sometimes increase and sometimes decrease. In both of these observations, the possible reason can be that we are not changing the difficulty while varying the transaction and transaction ratio. In Fig. 9 (a) and (b), with the varying number of transactions and transaction ratio the speedup is almost similar but with an increase in community members speedup is higher over the smaller community as well over serial.

Due to page limitation remaining results are presented in appendix as follows: Remaining results for transaction execution time and speedup in Appendix C. Block execution time (end-to-end time) at miner Appendix D. For varying number of transaction from 500 to 2500 Appendix E.

V. Conclusion

The main question we try to answer in this work is that, *is it possible to improve the throughput by adding distributed power collectively through mining pools in the current blockchain system?* Hence we proposed a distributed framework *DiPETrans* to execute transactions of block parallelly on multiple trusted nodes (part of the same community).

We tested our prototype on actual Ethereum transactions and achieved linear performance gain to the number of workers. We saw performance gain in the runtime of contract and monetary transaction. We tested *DiPETrans* on different workloads where number of transactions varies from 100 to 500 with varying contract transactions : non-contract transactions = 1:1, 1:2, 1:4, 1:8, 1:16. We found that with the increase in the number of transactions per block, speedup also increases in a distributed setting and helps in improving the throughput. We observe that if the number of contract call increases in a block execution time also increases. We also observed that speedup linearly increases with the increase in the size of the community. In literature, there is no such study done before, and that can be the novelty of this work.

As part of our future work, assuming the number of transactions will increase over time, we can think of a community to work on proposing multiple blocks parallelly. As suggested earlier, another way to improve performance is to use STM at slave nodes and parallelly execute the transactions using multi-cores instead of serial execution. In this work, we have assumed that there are no nested contract calls, but we can think of an approach to provide support for those transactions. Also, we can further improve the performance of mining by making use of all the cores available with the system and dividing the search space for POW based on that. Apart from the above optimization, we are also planning to adopt a distributed approach within the community instead of the master-slave approach.

REFERENCES

[1] Wef: bilding blockchains. http://www3.weforum.org/docs/WEF_Building-Blockchains.pdf. [Online; accessed 17-5-2019].

[2] R. Zhang, R. Xue, and L. Liu, "Security and Privacy on Blockchain," https://ui.adsabs.harvard.edu/abs/2019arXiv190307602Z, Tech. Rep., Mar 2019.

[3] Coindesk - leader in blockchain news. https://www.coindesk.com/. State of Blockchian - Q4 2017. (November 27 2017).

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.

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

[6] V. Saraph and M. Herlihy, "An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts," *CoRR*, vol. abs/1901.01376, 2019. [Online]. Available: http://arxiv.org/abs/1901.01376

[7] "Ethereum (ETH): open-source blockchain-based distributed computing platform," https://www.ethereum.org/, [Online; accessed 17-5-2019].

[8] Hyperledger: open source blockchain technologies. https://www.hyperledger.org/. [Online; accessed 17-5-2019].

[9] "Solidity documentation," https://solidity.readthedocs.io/en/v0.5.3/solidity-in-depth.html, [Online; accessed 26-3-2019].

[10] T. Max. What is proof-of-work (pow). https://medium.com/nakamo-to/what-is-proof-of-work-pow-2574ddebf916. [Online; accessed 17-5-2019].

[11] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, Jun. 2018.

[12] EOSIO: blockchain software architecture. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md. [Online; accessed 17-5-2019].

[13] Blockchain scalability. https://www.oreilly.com/ideas/blockchain-scalability. [Online; accessed 17-5-2019].

[14] I. Eyal, "The miner's dilemma," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 89–103.

[15] R. Qin, Y. Yuan, S. Wang, and F. Wang, "Economic issues in bitcoin mining and blockchain research," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, June 2018, pp. 268–273.

[16] Y. Lewenberg, Y. Bachrach, Y. Sompolinsky, A. Zohar, and J. S. Rosenschein, "Bitcoin mining pools: A cooperative game theoretic analysis," in *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '15. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2015, pp. 919–927.

[17] L. W. Cong, Z. He, and J. Li, "Decentralized mining in centralized pools," http://www.nber.org/papers/w25592, National Bureau of Economic Research, Working Paper 25592, February 2019.

[18] Mining pool. https://en.wikipedia.org/wiki/Mining_pool. [Online; accessed 17-5-2019].

[19] Alethio. Are miners centralized? a look into mining pools. https://media.consensys.net/are-miners-centralized-a-look-into-mining-pools-b594425411dc. [Online; accessed 17-5-2019].

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

[21] A. Zhang and K. Zhang, "Enabling concurrency on smart contracts using multiversion ordering," in *Web and Big Data*, Y. Cai, Y. Ishikawa, and J. Xu, Eds., Cham, 2018, pp. 425–439.

[22] P. S. Anjana, S. Kumari, S. Peri, and A. Somani, "Achieving greater concurrency in execution of smart contracts using object semantics," *CoRR*, vol. abs/1904.00358, 2019. [Online]. Available: http://arxiv.org/abs/1904.00358

[23] D. Chaum, "Blind signatures for untraceable payments," in *Advances in Cryptology*, D. Chaum, R. L. Rivest, and A. T. Sherman, Eds. Boston, MA: Springer US, 1983, pp. 199–203.

[24] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer, "Karma: A secure economic framework for peer-to-peer resource sharing," in *In Workshop on Economics of Peer-to-peer Systems*, vol. 35, 2003, pp. 919–927.

[25] B. Yang and H. Garcia-Molina, "Ppay: Micropayments for peer-to-peer systems," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 300–310.

[26] Multichain: open source blockchain platform. https://www.multichain.com/. [Online; accessed 17-5-2019].

[27] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 583–598.

[28] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 931–948.

[29] M. Bartoletti, L. Galletta, and M. Murgia, "A true concurrent model of smart contracts executions," *arXiv preprint arXiv:1905.04366*, 2019.

[30] L. Luu, Y. Velner, J. Teutsch, and P. Saxena, "Smartpool: Practical decentralized pooled mining," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1409–1426.

[31] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 17–30.

[32] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," *CoRR*, vol. abs/1708.03778, 2017. [Online]. Available: http://arxiv.org/abs/1708.03778

[33] Ethereum blockchain public dataset. https://bigquery.cloud.google.com/dataset/bigquery-public-data:ethereum_blockchain?pli=1. [Online; accessed 17-5-2019].

APPENDIX

This section is organized as follows:

Section No.	Section Name
A	Summary of Ethereum Transaction Data
B	Remaining Proposed Algorithms
C	Remaining Results and Observation for Transaction Execution Time and Speedup
D	Remaining Results and Observation for Block Execution Time (End-to-End Time) with Mining at Miner
E	Results and Analysis when Number of Transaction Varies from 500 to 2500 per Block

A. Summary of Ethereum Transaction Data

This section presents the data set designed for the experiment using extracted transaction from Ethereum blockchain and smart contract calls implemented in *C++* are given in the *Table I and Table II*. In the *Table I data-1-1-100* represents that 1:1 is the ratio of contractual : monetary transactions per block while 100 is the total number of transaction per block.

TABLE I: Summary of Simulated Transactions Data

S. No.	*Data (#Txns/Block)*	*#Blocks*	*#Contract_Txns*	*#Non_Contract_Txns*
1	data-1-1-100	3,880	193,959	194,000
2	data-1-1-200	1,940	193,959	194,000
3	data-1-1-300	1,294	193,959	194,100
4	data-1-1-400	970	193,959	194,000
5	data-1-1-500	776	193,959	194,000
6	data-1-2-100	5,705	193,959	376,530
7	data-1-2-200	2,895	193,959	385,035
8	data-1-2-300	1,940	193,959	388,000
9	data-1-2-400	1,448	193,959	385,168
10	data-1-2-500	1,162	193,959	386,946
11	data-1-4-100	9,698	193,959	775,840
12	data-1-4-200	4,849	193,959	775,840
13	data-1-4-300	3,233	193,959	775,840
14	data-1-4-400	2,425	193,959	776,000
15	data-1-4-500	1,940	193,959	776,000
16	data-1-8-100	16,164	193,959	1,422,432
17	data-1-8-200	8,434	193,959	1,492,818
18	data-1-8-300	5,705	193,959	1,517,530
19	data-1-8-400	4,311	193,959	1,530,405
20	data-1-8-500	3,464	193,959	1,538,016
21	data-1-16-100	32,327	193,959	3,038,738
22	data-1-16-200	16,164	193,959	3,038,832
23	data-1-16-300	10,776	193,959	3,038,832
24	data-1-16-400	8,082	193,959	3,038,832
25	data-1-16-500	6,466	193,959	3,039,020

TABLE II: Summary of Contract Transactions Calls

S.No.	*Fn Name*	*Fn Hash*	*Parameters*	*#Txns*	*Percentile*
1	transfer	0xa9059cbb	address, uint256	56,654	37.72
2	approve	0x095ea7b3	address, uint256	11,799	45.58
3	vote	0x0121b93f	uint256	11,509	53.24
4	submitTransaction	0xc6427474	address, uint256,bytes	8,163	58.67
5	issue	0x867904b4	address,uint256	5,723	62.49
6	__callback	0x38bbfa50	bytes32, string, bytes	5,006	65.82
7	playerRollDice	0xdc6dd152	uint256	4,997	69.15
8	multisend	0xad8733ca	address, address[],uint256[]	4,822	72.36
9	SmartAirdrop	0xa8faf6f0	-	4,467	75.33
10	PublicMine	0x87ccccb3	-	4,157	78.10
11	setGenesisAddress	0x0d571742	address, uint256, bytes	3,119	80.17

B. Remaining Proposed Algorithm

This section describes the proposed algorithms and a short description of them.

Algorithm 2: MasterMinerTask()

```
Data: ethereumData, slaveList, dataItemMap
Result: blockchain (blockList)
1  Procedure MasterMinerTask(ethereumData):
2      Block block;
3      for data ∈ ethereumData do
           // creates candidate block
4          block = CreateBlock(data);
5          if block.txnsList.size() > 0 then
               // Identify disjoint sets of txns
                  (Weakly Connected Components)
6              sendTxnsMap = Analyze(block.txnsList);
               // Parallel call to each slave to
                  assign transactions
7              for slave ∈ slaveList do
8                  ConnectSlave(slave, sendTxnsMap);
               // wait till all slave execution
                  completes
           // Start Block Mining
9          miningStatus = false;
10         for slave ∈ slaveList do
               // Parallel call to each slave to mine
                  block
11             SlaveMineBlock(slave, block);
12         while !miningStatus do
13             wait();
           // Broadcast the block for validation to
              all other peers and after more than 51%
              acceptance
14         blockchain.append(block);
```

Algorithm 2: MasterMinerTask() – The *MasterMiner* starts block creation by calling *CreateBlock()* (Algo. 4). The candidate block consists of block number, nonce, previous hash, miner detail (coin base address), transaction list, etc. The *Analyze()* (Algo. 1) is used to analyzes the candidate block transactions for sharding. The master receives a response *sendTxnsMap* from *Analyze()*, which consist a map of *slaveID* and transactions list. Each slave is allocated a unique id during the initialization of the master server. Master connects to the slave by creating parallel threads with a call to *ConnectSlave()* (Algo. 7). The master waits for the slave's to complete the transaction execution. So as soon as the master receives the *slave response (SResponse)*, it makes changes to its state in *dataItemMap*.

After transaction execution, the master starts mining (determine PoW) by setting *miningStatus* to *false*. For this master distributes task among the slaves to mine the block using asynchronous call to *SlaveMineBlock()* (Algo. 6). Master calls each slave to mine from different starting nonce in sequence. In this way, search space is distributed among the slaves for PoW calculation. The master waits until the *miningStatus* turned to be true by a slave. The slave who finds the correct PoW sends the information to the master. Finally, master saves the nonce and broadcast the block in the network for validation.

Algorithm 3: DefaultValidator() – When validator receives a block for the validation, it re-executes the transactions and match the final state of the *dataItemMap* state. Since in this approach dependency (shard) information is not added to the block by the miner, the master validator needs to call the

Analyze() to determine the disjoint sets of transactions. Then master validator follows the same approach as the miner to distribute transactions to the slave and waits for all slaves to complete. The master validator does not have to mine the block. The master validator updates its *dataItemMap* state based on the responses from the slaves. Finally, it verifies its *dataItemMap* state with the block's *dataItemMap* state. If *dataItemMap* state matches after successful execution of block's transaction, then it checks for PoW by verifying $hash(block) < difficulty$. If both the checks come out to be successful, an acceptance message propagated in the network. Otherwise, the block is rejected by the validator, and no propagation can and can't be entertained. After majority acceptance or consensus, the block is added to the *blockchain*.

Algorithm 3: DefaultValidator()

```
   Data: block
   Result: blocklist
1  Procedure MasterValidator(block):
2      if block.txnsList.size() > 0 then
           // Identify disjoint sets of txns (Weakly
              Connected Components)
3          sendTxnsMap = Analyze(block.txnsList);
           // Parallel call to each slave to assign
              transactions
4          for slave ∈ slaveList do
5              ConnectSlave(slave, sendTxnsMap);
           // wait till all slave execution completes
6          reject = false;
           // Verify PoW
7          if hash(block) > difficulty then
               // PoW is incorrect
8              reject = true;
9          else
               // Verify dataItemMap with block's
                  dataItemMap state.
10             for adr, value ∈ block.dataItemMap do
11                 if dataItemMap[adr] != value then
12                     reject = true;
13                     break;
       // Accepted by more than 51% peers
14     if !reject then
15         blockchain.append(block);
```

***SharingValidator* () –** In this function sharding information is added in the block by the miner, so the master validator need not to call the *Analyze*() at *line 3* in Algo. 3. Therefore the overhead caused by Analyze function can be avoided, and the validator utilizes the analysis work done by the miner for parallel execution. So, in the *Sharing Validator*, the *MasterValidator* deterministically assign the different shards based on the dependency information in the block to the different *SlaveValidator* along with the current state of the data items from its local chain for transaction execution. The rest of the functionality of this algorithm is same as *Default Validator*.

***Algorithm* 4: *CreateBlock()* –** In this function, the master creates the candidate block by picking pending transactions from pending transactions pool. Master also assigns block number, previous hash, and miner (coin base address) to the block. There can be some uncles blocks which are valid blocks and the miner who proposed that blocks deserve incentive for their work. If these blocks are added by the upcoming blocks (< 8), also called nephew blocks, they are given partial incentive based on the below formula. And some incentive is also given to the miner who adds the uncle blocks.

$uncleMinerReward = ((uncleBlockNumber + 8 - nephewBlockNumber) * baseReward)/8$

$nephewMinerReward = (baseReward)/32$

The maximum limit on the inclusion of uncle blocks is 2.

Algorithm 4: CreateBlock()

```
   Data: data, prevHash
   Result: block
1  Procedure CreateBlock(data, prevHash):
2      Block block;
3      block.number = data.number;
4      block.prevHash = prevHash;
5      block.miner = data.miner;
       // creates candidate block
6      for tx ∈ data.txns do
7          Transaction txn(tx.txID, tx.to, tx.from, tx.value, tx.input,
             tx.creates);
8          block.txnsList.append(txn);
9      for u ∈ data.uncles do
10         Uncle unc(u.number,u.miner);
11         block.unclesList.append(unc);
```

***Algorithm* 5: *MiningStatus()* –** This function is called by a slave to send the nonce value for the block to master and signaling master that mining has complete. In this function, slave checks for block number on which master is working on with its block number and the *miningStatus* not true. Then, the slave updates the nonce value of the block and set *miningStatus* to true. Master notices change in *miningStatus* variable and come out of the wait loop. And starts working on the next block after sending the current block for its inclusion in blockchain and verification by validators.

Algorithm 5: MiningStatus()

```
   Data: block, nonce, number
   Result: block
1  Procedure MiningStatus(nonce, number):
2      if block.number == number && !miningStatus then
3          block.nonce = nonce;
4          miningStatus = true;
```

***Algorithm* 6: *SlaveMineBlock()* –** In this function, slaves receive the starting nonce and interval along with the block to find the PoW. The PoW is found when $hash(block) < difficulty$ is found for a particular value of nonce. Otherwise, a nonce is incremented by interval to try again in an infinite loop. The difficulty we have set for now takes approximately 15 seconds to mine a block, which is close to the current average time of Ethereum block execution. The actual difficulty is much larger than what we are using, considering the resources deployed by miners to find PoW.

***Algorithm* 7: *ConnectSlave()* –** Master calls *ConnectSlave()* (Algo. 7) using parallel threads to assign different

Algorithm 6: SlaveMineBlock()

```
Data: block, nonce, interval, difficulty
Result: nonce
Procedure SlaveMineBlock(block, nonce, interval):
    while true do
        if hash256(block) < difficulty then
            MiningStatus(nonce,block.number);
            break;
        nonce += interval;
```

task to the slaves. Here master identifies each slave *local-DataItemMap* state from it's (master's) *dataItemMap* state based on the transactions a slave is going to execute. Master sends transaction list (shards) and associated *dataItemMap* to each slave by calling *SlaveRecvTxns()* (Algo. 8). On this call slave executes the transactions while master waits for the slaves responses (*SResponse*). The *SResponse* is used to update the master's *dataItemMap*.

Algorithm 7: ConnectSlave()

```
Data: slave, sendTxnsMap, dataItemMap
Result: dataItemMap
Procedure ConnectSlave(slave, sendTxnsMap):
    // Identify Associated dataItemMap for each
       slave
    for tx ∈ sendTxnsMap[SID] do
        localDataItemMap[tx] = dataItemMap[tx] ;
    // send txns to slave to execute, receive
       updated state
    SResponse = SlaveRecvTxns(sendTxnsMap[slave],
      localDataItemMap);
    // update master's dataItemMap state
    for adr, dataItem ∈ SResponse.dataItemMap do
        dataItemMap[adr] = dataItem;
```

Algorithm 8: SlaveRecvTxns() – While executing *SlaveRecvTxns()* (i.e., Algo. 8), the slave receives the transaction list and associated *dataItemMap* state from the master. The slave first identifies smart contract and non-smart contract calls (transactions). If the transaction is a smart contract call, then the transaction is executed by invoking *CallContract()* to execute respective smart contract method. Otherwise, non-smart contract calls are considered as monetary exchanges and executed within the scope of this function. For *CallContract()* transaction execution, we have implemented top 11 functions calls in Ethereum which cover 80% of real transactions of block numbers between 4370000 and 4450000.

C. Remaining Results and Observation for Transaction Execution Time and Speedup

This section presents the experimental analysis done for transaction execution time for *Workload-3* and speedup achieved in transaction execution by parallel miner and validator over serial. First, we present the analysis for transaction execution time for *Workload-3*. Then we present the analysis for the remaining analysis of the speedup for *Workload-1*, *Workload-2*, and *Workload-3*.

Algorithm 8: SlaveRecvTxns()

```
Data: txnsList, dataItemMap
Result: SResponse
Procedure SlaveRecvTxns(txnsList, dataItemMap):
    // Execute txns serially
    for tx ∈ txnsList do
        // Smart Contract txn
        if tx is contractCall then
            CallContract (tx);
        // Non-Smart Contract txn
        else if tx.value ≤ dataItemMap[tx.from].value then
            dataItemMap[tx.fromAddress].value -= tx.value;
            dataItemMap[tx.toAddress].value += tx.value;
        else
            // Invalid txn: txn execution failed!
```

1) Transaction Execution Time Analysis: **Workload-3:** Fig. 10 shows the analysis for a fixed number of transactions (500) per block with varying community size. This workload is designed to see how the transaction ratio (contract call: monetary transaction) in each block will have an impact on the performance. We can observe in Fig. 10(a), 10(b), 10(c), and *10(d)* that 1 slave is performing worst due to the overhead of static analysis and communication with master. Other slave configurations from 2 to 5 are all outperforming over serial, and execution time decreases as the number of slaves increases. Also, the smaller the number of contractual transaction per block the performance will be the better. This is because of the external method call by the contractual transaction. Similar to *Workload-1* and *Workload-2*, Fig. 10(d) shows that there is no much performance difference in *Sharing Validator* and *Default Validator*.

2) Speedup Analysis: Here we will present the result analysis for all three workloads based on speedup achieved by parallel miner and validator over serial miner and validator.

Workload-1: *Fig. 6* shows the average transactions speedup achieved by the miner (without mining) and validator. As shown in the Fig. 6(a), 6(b), 6(c), and *6(d)* the mean speedup increases as the number of transactions per block increases. Also, one slave is performing worst due to the small overhead of static analysis and communication with the master. Other slave configurations from 2 to 5 are all working better than serial. Fig. 4(d) shows the line plot comparison for average speedup over serial by *Default Validator* and *Sharing Validator*. The only difference between these two validators is that *Default Validator* needs to run static analysis on transactions present in a block before execution. The *Default Validator* is supposed to take more time compared to *Sharing Validator*. But the experiment shows no significant benefits of information sharing as the time taken by static analysis function takes very less time. However, when the number of transaction per block increases further, then possibly, the benefits of information sharing can be observed.

Workload-2: In Fig. 7(a), 7(b), and *7(c)*, we can observe that when the number of contract transactions decreases per block, the overall speedup increases because contractual transaction includes the external contract method calls. Also, we

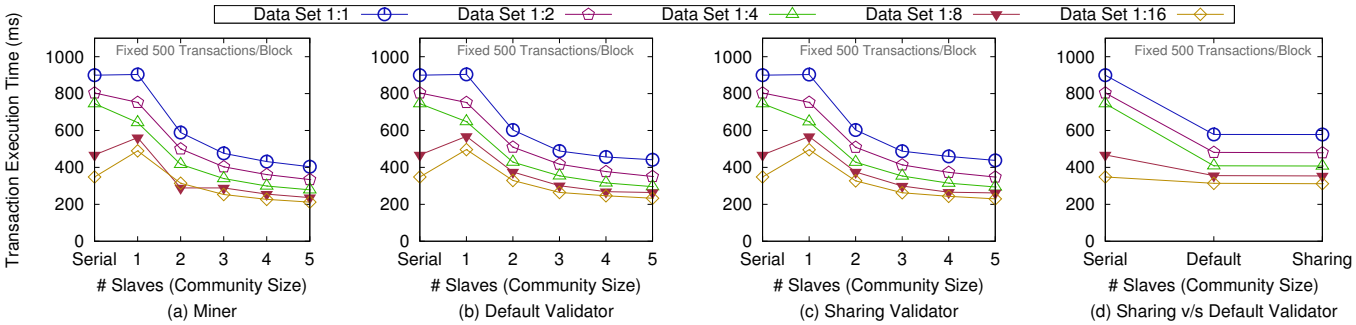

Fig. 10: Workload-3: Average Transaction Execution Time by Miner (Without Mining) and Validator

can see that the speedup increases till 1:4 (contract : monetary transactions) and then decreases with a further decrease in the number of contract transactions per block. Fig. 7(d) shows the line plot comparison for speedup over serial by *Default Validator* and *Sharing Validator*. The experiment shows there is no significant time taken by analyzing function to gain some performance improvement over *Sharing Validator*. Hence both *Default Validator* and *Sharing Validator* are performing almost same.

Workload-3: As we can see in all this Fig. 11(a), 11(b), 11(c), and *11(d)* that 1 slave is performing worst due to the obvious reason of overhead of static analysis and communication with slave. Other slave configurations all outperforming over serial. Also, the smaller the number of contractual transaction per block the performance will be the better. However Fig. 11(d) shows that there is no much performance benefit due to information sharing with the validator.

Below tables shows the respective numbers for the workloads. Table III and Table IV shows the transaction execution time while Table V and Table VI the respective speedup.

TABLE III: Workload-1: Average Transaction Execution Time (ms) taken by Miner (Without Mining) and Validators

Workload 1: Execution Time-Averaged Across Data Set							
Average Transaction Execution Time (ms)			Number of Transactions/Block				
			100	200	300	400	500
Number of Slaves	Serial	Miner	209.178	259.680	333.517	485.548	652.948
		Validator	209.178	259.680	333.517	485.548	652.948
	1 Slave	Miner	214.934	338.606	449.450	560.487	670.041
		No Info Validator	222.632	345.584	455.267	564.490	673.896
		Info Validator	222.769	345.301	454.884	564.661	673.295
	2 Slave	Miner	163.906	239.518	304.819	372.045	435.397
		No Info Validator	175.435	250.367	315.849	384.773	449.132
		Info Validator	175.397	250.078	315.076	383.825	447.844
	3 Slave	Miner	145.597	202.979	255.478	305.016	352.256
		No Info Validator	159.253	218.479	270.447	320.623	365.106
		Info Validator	159.765	218.633	269.393	319.897	363.379
	4 Slave	Miner	138.346	188.285	231.101	272.976	314.880
		No Info Validator	154.815	204.332	248.875	289.148	332.949
		Info Validator	154.416	203.515	247.842	287.508	331.248
	5 Slave	Miner	137.786	181.983	219.573	255.971	292.882
		No Info Validator	153.819	198.089	236.746	273.503	316.972
		Info Validator	155.583	196.923	235.712	270.301	314.148

D. Block Execution Time (End-to-End Time) with Mining at Miner

This section presents the analysis for the end-to-end block creation time, including mining (PoW) at the miner for transaction varies from 100 to 500 in *Workload-1* while it is fixed to 500 in *Workload-2* and *Workload-3* however other parameters as data set and community size varies respectively.

TABLE IV: Workload-2 and Workload-3: Average Transaction Execution Time (ms) taken by Miner (Without Mining) and Validators for Fixed 500 Transactions/Block

Workload-2 and Workload-3: for Fixed 500 Transactions/Block							
Average Transaction Execution Time (ms)			Data Set (Contractual:Monetary Transactions)				
			1:1	1:2	1:4	1:8	1:16
Number of Slaves	Serial	Miner	899.501	803.577	745.571	467.861	348.230
		Validator	899.501	803.577	745.571	467.861	348.230
	1 Slave	Miner	903.435	752.360	643.278	561.328	489.806
		No Info Validator	904.015	752.003	648.402	568.377	496.685
		Info Validator	903.008	752.794	647.327	567.520	495.829
	2 Slave	Miner	588.783	500.214	416.558	357.128	314.299
		No Info Validator	602.825	509.827	428.580	375.175	329.252
		Info Validator	602.512	508.785	428.127	372.610	327.189
	3 Slave	Miner	476.162	403.198	339.778	288.770	253.375
		No Info Validator	488.707	417.246	354.410	300.697	264.469
		Info Validator	487.988	413.735	353.356	299.257	262.558
	4 Slave	Miner	431.591	361.182	299.185	255.518	226.925
		No Info Validator	456.401	377.233	316.041	268.871	246.201
		Info Validator	459.720	372.981	314.459	265.778	243.300
	5 Slave	Miner	403.518	334.074	278.474	236.432	211.909
		No Info Validator	441.508	350.673	295.938	263.882	232.858
		Info Validator	437.905	347.662	293.810	261.896	229.465

TABLE V: Workload-1: Average Speedup by Parallel Miner (Without Mining) and Validator

Workload-1: Execution Time-Averaged Across Data Set							
Average Speedup			Number of Transactions/Block				
			100	200	300	400	500
Number of Slaves	Serial	Miner	1	1	1	1	1
		Validator	1	1	1	1	1
	1 Slave	Miner	0.869	0.723	0.723	0.843	0.953
		No Info Validator	0.838	0.708	0.713	0.837	0.947
		Info Validator	0.837	0.709	0.714	0.837	0.948
	2 Slave	Miner	1.148	1.024	1.068	1.271	1.467
		No Info Validator	1.069	0.979	1.029	1.230	1.421
		Info Validator	1.067	0.980	1.031	1.233	1.425
	3 Slave	Miner	1.295	1.209	1.273	1.550	1.813
		No Info Validator	1.179	1.121	1.200	1.475	1.747
		Info Validator	1.174	1.119	1.205	1.479	1.755
	4 Slave	Miner	1.371	1.303	1.406	1.734	2.032
		No Info Validator	1.217	1.198	1.305	1.636	1.921
		Info Validator	1.220	1.202	1.311	1.648	1.933
	5 Slave	Miner	1.388	1.350	1.478	1.848	2.185
		No Info Validator	1.249	1.237	1.370	1.728	2.022
		Info Validator	1.244	1.243	1.376	1.747	2.040

1) Transaction Execution Time Analysis: Fig. 8(a) shows the line plots for mean end-to-end block creation time taken by the miner (with mining) for *Workload-1*. The overhead of static analysis and communication is negligible with mining. Here, all slaves configuration are performing better than serial. Since the PoW is random nonce for which the hash of a block is less than the given difficulty, it can take a variable amount

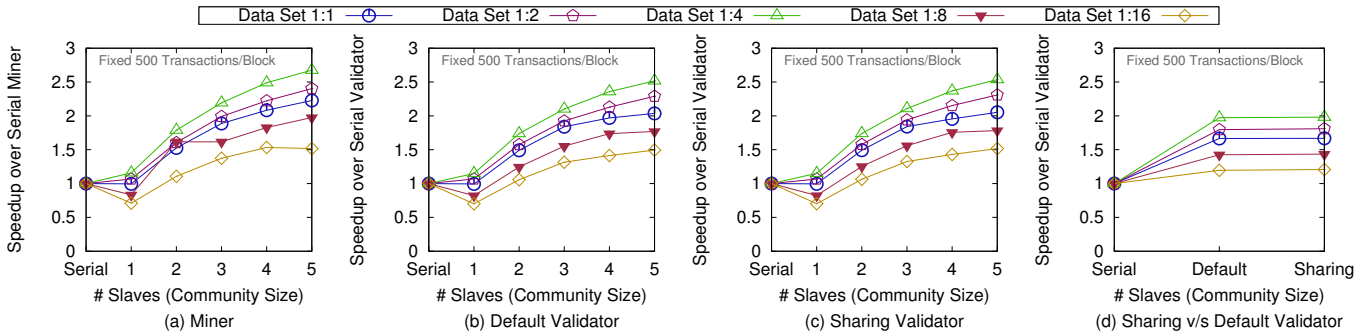

Fig. 11: Workload-3: Average Speedup by Miner (Without Mining) and Validator for Transaction Execution

TABLE VI: Workload-2 and Workload-3: Average Speedup by Parallel Miner (Without Mining) and Validators

Workload-2 and Workload-3: for Fixed 500 Transactions/Block							
Average Speedup			Data Set (Contractual:Monetary Transaction)				
			1:1	1:2	1:4	1:8	1:16
Number of Slaves	Serial	Miner	1	1	1	1	1
		Validator	1	1	1	1	1
	1 Slave	Miner	0.995	1.067	1.158	0.832	0.710
		No Info Validator	0.994	1.068	1.149	0.822	0.701
		Info Validator	0.996	1.067	1.151	0.823	0.702
	2 Slave	Miner	1.527	1.605	1.788	1.308	1.107
		No Info Validator	1.491	1.575	1.738	1.245	1.057
		Info Validator	1.492	1.578	1.740	1.253	1.064
	3 Slave	Miner	1.888	1.992	2.193	1.617	1.373
		No Info Validator	1.840	1.925	2.102	1.553	1.316
		Info Validator	1.842	1.941	2.108	1.561	1.325
	4 Slave	Miner	2.083	2.223	2.490	1.828	1.534
		No Info Validator	1.970	2.129	2.357	1.737	1.413
		Info Validator	1.956	2.153	2.369	1.757	1.430
	5 Slave	Miner	2.228	2.404	2.675	1.975	1.642
		No Info Validator	2.036	2.290	2.517	1.770	1.494
		Info Validator	2.053	2.310	2.536	1.783	1.517

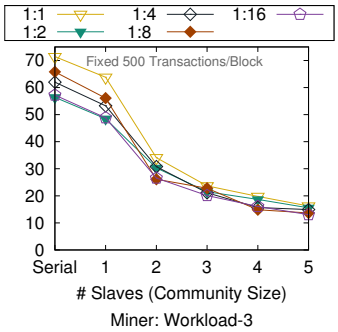

Fig. 12: Workload3: Average End-to-End Block Creation Time by Miner with Mining (PoW)

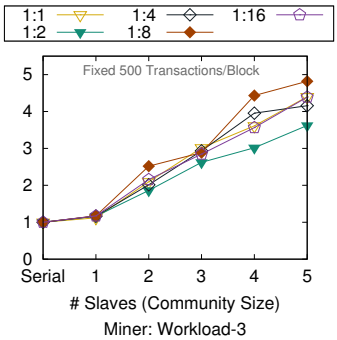

Fig. 13: Workload3: Average End-to-End Block Creation Speedup by Parallel Miner over Serial Miner with Mining (PoW)

of time to find nonce. Also, the serial and slave configurations have a different order of transactions in the final block. Both blocks are correct as proposed by the miner and considered as the final order of transaction execution. It is possible that due to some outliers in serial execution resulted higher mean for the end-to-end time required to create a block as compared to 1 slave. Another observation is that the time required to create block increases linearly as the number of transactions per block increases. Across the different number of transactions, the trend remains consistent with slaves, the higher number of slaves takes less time than the less number of slaves.

For *Workload-2* results are shown in Fig. 8(b). The reason for serial and one slave for these plots remains the same as explained above. The time required to create block across different data sets varies and does not show any pattern based on a contract to monetary transactions ratio. Since it largely depends on the PoW search. With PoW, we are always guaranteeing that when the number of slaves increases, it will take less time to create a block (with mining) than the serial.

Similarly, in Workload-3 when the number of transactions per block is fixed to 500 and community size increases (i.e., slaves in the community increases) the time taken to mine a block always takes lesser time than the serial and smaller size community. Fig. 12 confirms this observation; however, it is challenging to claim about which data set is doing better over others, and the reason is that mining time dominates the transaction execution time.

2) Speedup Analysis: For *Workload-1* Fig. 9(a) shows the mean speedup achieved by parallel community-based miner over serial by the miner (with mining). Here, all slaves configuration are achieving better speedup over serial. The observation here is that the speedup varies as the number of transactions per block increases. Across the different number of transactions, the trend remains consistent with slaves, the higher number of slaves gives higher speed up than the less number of slaves.

Fig. 9(b) shows the line plots for mean speedup achieved over serial by the parallel miner with mining for *Workload-2*. The time required to create block across different data sets varies and does not show any pattern based on transactions ratio (i.e., contractual : momentary transactions). Since it largely depends on the PoW search. With PoW, we are always

guaranteeing that more number of slaves gives higher speedup over serial with mining. In *Workload-3* the speedup increases with increase in the size of the community, but there are no fixed trained with varying transaction ratio.

E. Results and Analysis when Number of Transaction Varies from 500 to 2500 per Block

This section includes the experiment done on a varying number of transactions from 500 to 2500 per block. Similar to the earlier experiments where the number of transactions per block varies from 100 to 500, here in this experiments for *Workload-1* number of transactions vary from 500 to 2500 and transaction execution time, and the speedup is averaged over varying data sets (i.e., from 1:1 to 1:16). In *Workload-2* data set varies from 1:1 to 1:16 while the number of transactions per block remains fixed to 2500. While in *Workload-3* community size varies from 1 slave up to 5 slaves in the community, and the number of transactions per block remains fixed to 2500. In all the figures, serial execution served as a baseline.

1) Transaction Execution Time Analysis: **Workload-1:** Fig. 14 shows the average transactions execution time taken by the miner and validator. As shown in the Fig. 14(a), Fig. 14(b), Fig. 14(c), and Fig. 14(d) the time required to execute transactions per block increases as the number of transactions increases in a block. Also, the 1 slave is performing worst due to the overhead of static analysis and communication with the master. Other slave configurations from 2 to 5 all are better than serial. Fig. 14(d) shows the mean transactions execution time taken by *Default Validator*, *Sharing Validator* and Serial. The only difference between these two validators is that *Default Validator* needs to run static analysis on block transactions before execution. The *Default Validator* is supposed to take more time compared to *Sharing Validator*. The experiment shows slight performance improvement for *Sharing Validator* over *Default Validator* as the execution time and is significant enough to see the difference.

Workload-2: In this workload, the number of transactions per block is fixed to 2500 while the data set varies from 1:1 to 1:16. In Fig. 15(a), 15(b), and *15(c)*, it can be seen that decreasing the number of contract transactions than monetary transaction per block the overall time required to execute transactions also decreases because contractual transaction includes the external calls. Further, it can be noticed that serial execution outperforms 1 slave configuration due to the static analysis and communication overhead associated with one slave configuration. But other configurations from 2 to 5 slaves per community outperform over serial with the increases in the number of slaves in the community. But with the increase of monetary transaction in the block, serial execution started giving better performance because it may be because of communication dominates the transaction execution. In this figures, we can see that the time required to execute more transactions per block decreases as the number of contract transactions decreases. Fig. 15(d) shows that the parallel validator is always taking less time than serial, and we can also observe a significant gap as we increase the number of monetary transactions per block. The *Sharing Validator* achieved a slight improvement over *Default Validator* infect they are very close to each other.

Workload-3: Fig. 16 shows the analysis for a fixed number of transactions (2500) per block with varying community size. Here we see how the transaction ratio in each block will have an impact on the performance. We can observe in Fig. 16(a), 16(b), 16(c), and 16(d) that 1 slave is performing worst due to the overhead of static analysis and communication with master. Other slave configurations from 2 to 5 are all working better than serial, and execution time decreases as the number of slaves increases. Also, the smaller the number of contractual transaction per block the performance will be the better, i.e., 1:1 is taking higher time then 1:2 and 1:16 is taking least time among them since it consist 16$\times$ more monetary transactions than contractual transactions in a block. In Fig. 16(d) we can see the slight performance difference in *Sharing Validator* and *Default Validator*.

2) Speedup Analysis: **Workload-1:** Fig. 17 shows the mean speedup obtained by the parallel miner (without mining) and validator over serial miner and validator.

As shown in the Fig. 17 the mean speedup increases as the number of transactions per block increases but, the serial is outperforming 1 slave configuration of community-based parallel execution. This happens due to the static analysis and communication overhead associated with master and one slave communicate with the master. Other settings, i.e., 2 to 5 slaves in the community, all achieving better speedup over serial. Also, there is a drop in speedup going from 500 to 1000, but then onwards there is a steady increase in speedup. Next *Default Validator* and *Sharing Validator* is outperforming over serial which can be seen in Fig. 14(d).

Workload-2: In this workload, we fixed the number of transactions per block to 2500. However, the data set varies from 1:1 to 1:16, i.e., contractual to monetary transaction ratio varies. In Fig. 18(a), 18(b), and *18(c)*, it can be observed that by varying the ratio of contractual to monetary transaction the overall speedup increase because contractual transaction drops with number of increase in monetary transaction per block. Further, we can observe that speedup increases till 1:8 and then decreases on the further decrease in the number of contract transactions per block. Fig. 18(d) shows there is a slight performance improvement in *Sharing Validator* over *Default Validator*.

Workload-3: Fig. 19 shows that 1 slave is performing worst due to the overhead of static analysis and communication. Other slave configurations from 2 to 5 are all doing better than serial, and speedup increases as the number of slaves increases. Also, the smaller the number of contractual transaction per block the performance will be the better. As explained in the *Workload-2*, this is because of the external method call by the contractual transaction.

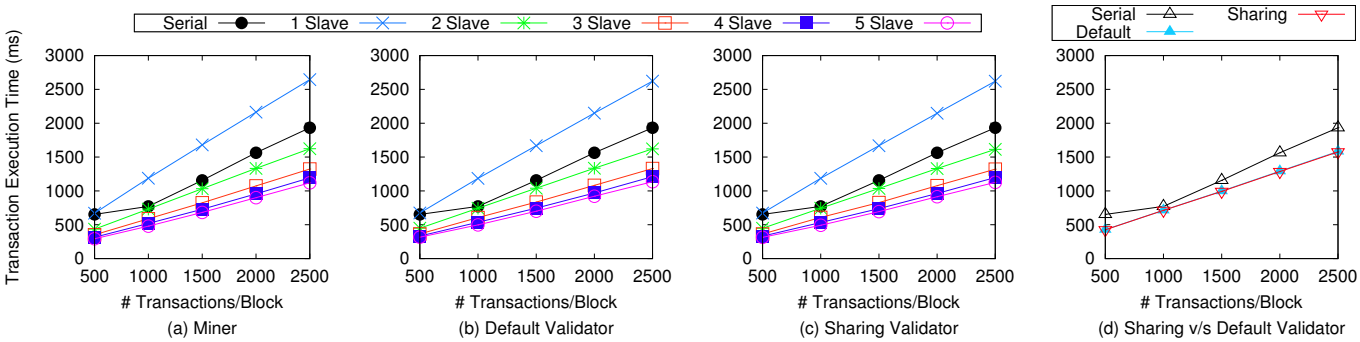

Fig. 14: Workload 1: Average Transaction Execution Time by Miner (without Mining) and Validator when Transactions Varies from 500 to 2500

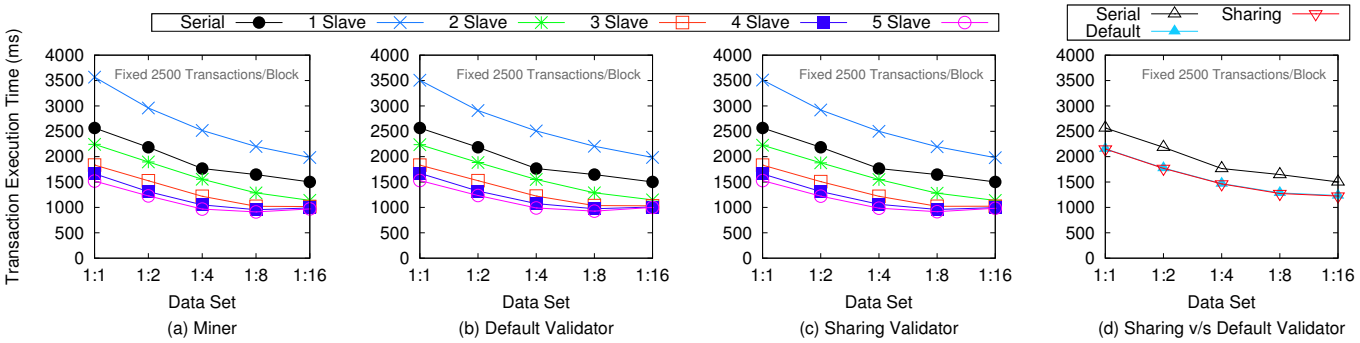

Fig. 15: Workload 2: Average Transaction Execution Time by Miner (without Mining) and Validator for 2500 Transactions/Block

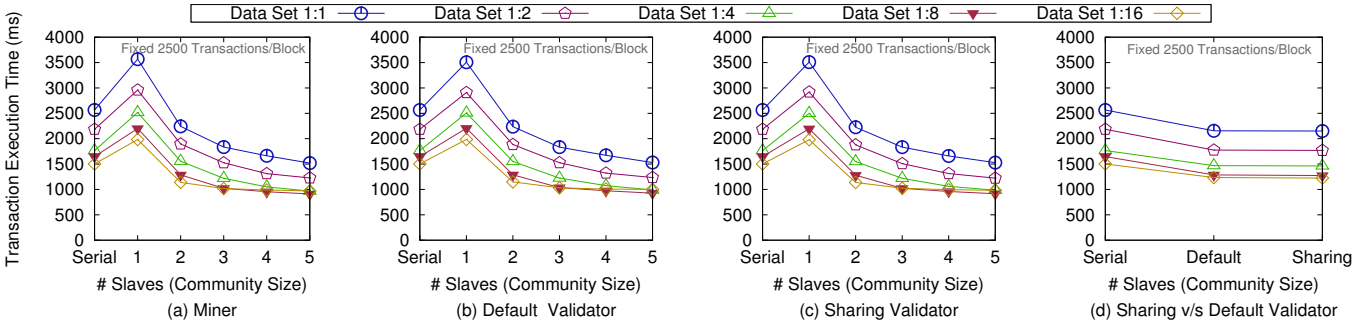

Fig. 16: Workload 3: Average Transaction Execution Time by Miner (without Mining) and Validator for 2500 Transactions/Block

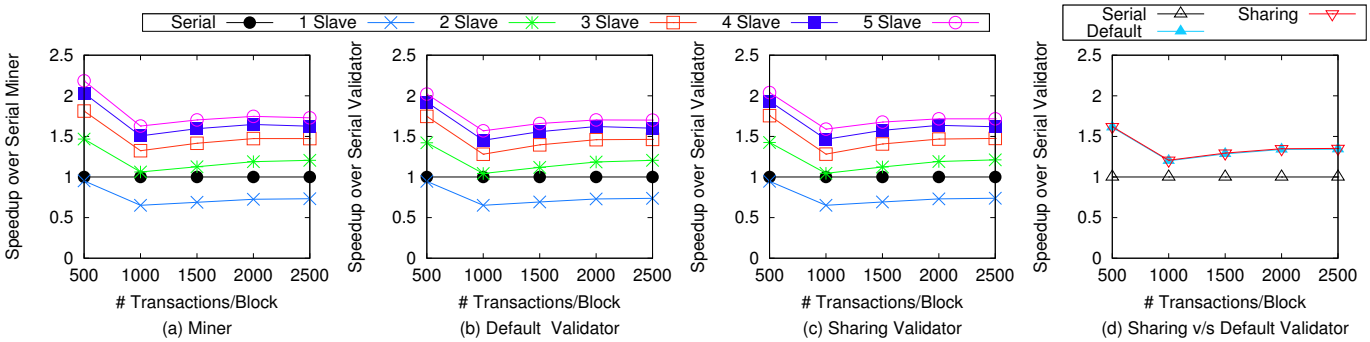

Fig. 17: Workload 1: Average Speedup by Miner (without Mining) and Validator for Transaction Execution when Transactions Varies from 500 to 2500

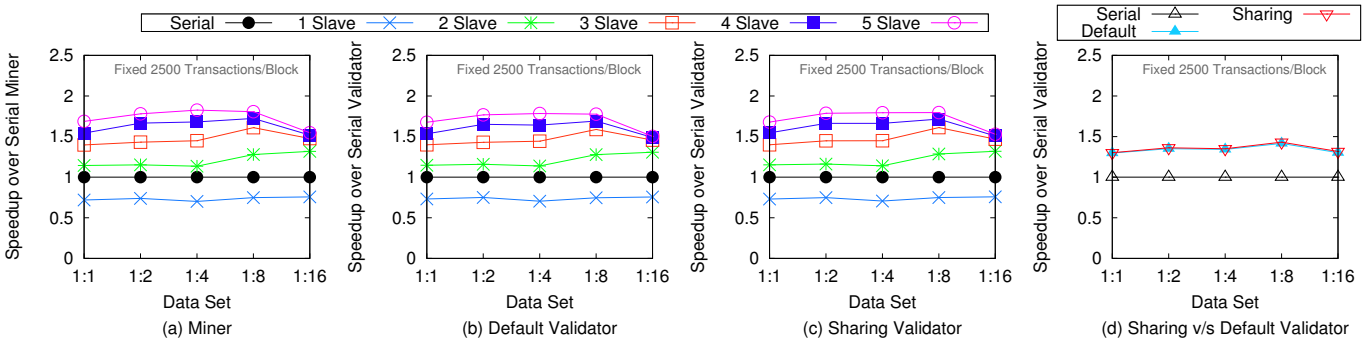

Fig. 18: Workload 2: Average Speedup by Miner (without Mining) and Validator for Transaction Execution for 2500 Transactions/Block

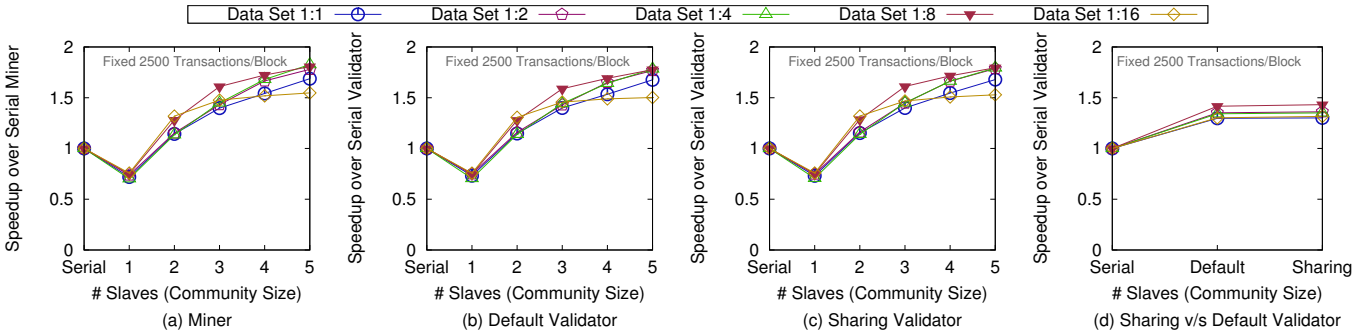

Fig. 19: Workload 3: Average Speedup by Miner (without Mining) and Validator for Transaction Execution for 2500 Transactions/Block