BlockPilot: A Proposer-Validator Parallel Execution Framework for Blockchain

Haowen Zhang
zhanghaowen@mail.ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, China

Jing Li
lj@ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, China

He Zhao
zhaoh@hfcas.ac.cn
Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, Anhui, China

Tong Zhou
tzhou@hfcas.ac.cn
Hefei Institutes of Physical Science, Chinese Academy of Sciences
Hefei, Anhui, China

Nianzu Sheng
nianzu@mail.ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, China

Hengyu Pan
hypan@mail.ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, China

ABSTRACT

Traditional blockchain systems still struggle with limited throughput, particularly those compatible with EVM, which are crucial in many blockchain applications. One of the main reasons arises from serial execution, which doesn't exploit parallelism in transaction execution. Although some recent literature introduced concurrency control mechanisms to execute transactions in parallel, they do not work efficiently in real-world blockchains where proposers and validators have different execution contexts, which means varying execution deterministic levels and execution quantities.

In this work, we propose BlockPilot, an overall parallel execution framework for blockchains. In the framework, proposers and validators can execute transactions concurrently with different deterministic levels. To enable parallel execution and produce a serializable schedule in proposers, we introduce an Optimistic Concurrency Control (OCC) algorithm called OCC-WSI. Meanwhile, validators require scheduling conflicting transactions to support parallel execution and commit transactions in the same schedule as proposers. Given that validators process more blocks than proposers, a parallel and pipeline workflow is designed for validators to process multiple blocks concurrently. We implement a prototype system based on Go Ethereum and evaluate its performance using real-world blocks. Our experimental results demonstrate that BlockPilot can accelerate the processing of a single block by 3.18x and achieve a maximum speedup of 7.71x in processing multiple blocks in 16 threads for validators, as well as a 4.89x acceleration for proposers compared to Go Ethereum.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent computing methodologies**; • **Software and its engineering** → *Software performance.*

KEYWORDS

blockchain, smart contract, parallel execution, optimistic concurrency, deterministic concurrency, pipeline

ACM Reference Format:
Haowen Zhang, Jing Li, He Zhao, Tong Zhou, Nianzu Sheng, and Hengyu Pan. 2023. BlockPilot: A Proposer-Validator Parallel Execution Framework for Blockchain. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605621

1 INTRODUCTION

Blockchain technologies have received increasing attention with the growing popularity of cryptocurrencies [25, 30]. The introduction of programmable smart contracts [14] in Ethereum further expanded the applications of blockchain beyond cryptocurrencies. However, the limited throughput of blockchain remains a serious problem, hindering their wider adoption.

The throughput of blockchain systems is defined as the number of transactions executed per second and is primarily influenced by two factors: block interval and block size. Block interval represents the average time taken to generate a new block. To shorten the block interval, researchers have proposed various consensus protocols [21, 22]. For example, Conflux [22] can generate a block in less than 0.1 seconds. Nevertheless, improving consensus protocols alone is not sufficient in improving the overall performance of the blockchain since the block size relies on the number of transactions executed within a block interval.

Blockchains that implement the account model process transactions serially to reach a consensus on the sequence of transactions. Although the model is simple to implement, it does not support multiple threads on multi-core processors. Thus, executing transactions in parallel is a promising way to accelerate transaction execution. Many researchers are currently focusing on parallel execution in Ethereum due to the increased complexity resulting from the introduction of smart contracts[3, 17, 23, 24, 27]. Recent works have adopted traditional ways, such as *Optimistic Concurrency Control* (OCC). To achieve OCC, some researchers [17, 23, 27] use a scheduler that re-executes conflicting transactions serially. Other studies [32] utilize Multi-Version Concurrency Control (MVCC) to allow for parallel execution. However, these studies fail to distinguish different block execution contexts in blockchain.

Transaction execution in blockchain has two specific contexts: proposing and validation[10]. In the proposing context, one or several nodes (*proposers*) execute transactions to propose a new block. Proposers then suggest appending a new block to the blockchain based on their consensus protocol by broadcasting their proposed blocks. In the validation context, other nodes (*validators*) receive the broadcasted block and verify its contents through the re-execution of transactions in the block. Executing transactions in different contexts may result in varying levels of execution determinism and transaction quantities, as explained below.

The parallel execution of transactions implies executing with different levels of determinism. Proposers can execute transactions in parallel as long as the execution result can be ordered in a *serializable* schedule [26]. Then, the proposers combine transactions into a block according to the schedule, where the observable results of the parallel execution are equivalent to those of some serial execution. On the other hand, validators must execute transactions in the block corresponding to a specific serial schedule to ensure the final execution outcome is equivalent across different nodes, even though the actual schedule may diverge.

Furthermore, in a Byzantium network, validators usually process more blocks than proposers due to the distributed consensus [19]. Different proposers may produce different blocks at the same height and broadcast them to the network. As a result, validators receive multiple blocks and need to execute transactions more efficiently to maintain a high throughput of blockchain.

Building on the insights above, this paper proposes an overall framework, BlockPilot, that enables both proposers and validators to execute transactions in parallel. In this framework, proposers can produce a block with a serializable schedule, while validators have to execute transactions in the same schedule as proposers. For proposers, we have designed a *Write Snapshot Isolation*-based OCC algorithm (*OCC-WSI*) to execute transactions in parallel and pack transactions into a block according to their serial schedule. For validators, we schedule transactions in the block to ensure the serial order and execute transactions in different threads according to the schedule. To execute transactions more efficiently in validators, A pipeline workflow for validators is designed to process multiple blocks to reduce the overall latency in transaction execution.

We have implemented a prototype system based on Go Ethereum (Geth) [15] and conducted a comprehensive evaluation of our prototype system. The evaluation results show that the system can achieve an average speedup of 3.18x on real-world Ethereum blocks for validators and 4.89x for proposers. Furthermore, if it executes multiple blocks in the system, the speedup can reach a maximum of 7.71x speedup. The main contributions of the paper are summarized as follows.

- An overall proposer-validator parallel execution framework, BlockPilot, for blockchain.
- An OCC-WSI parallel transaction execution algorithm for proposers.
- A pipeline workflow for validators to process multiple blocks and execute transactions in parallel.
- A prototype system implemented on Go Ethereum [15] and evaluate results using real-world Ethereum blocks.

2 BACKGROUND

2.1 Blockchain

A blockchain [25] is a distributed database that enables mutually untrusting nodes to maintain identical records by running a replicated state machine [28] in a chain of blocks. In this model, nodes (proposers) first execute transactions from the network and pack them into a block. Afterward, the block is broadcast to the network, and other nodes (validators) validate the block by re-executing transactions in it and add the valid block to the ledger.

Ethereum [30] extends the concept of blockchain by introducing support for Turing-complete programming framework in Solidity[13]. As a result, developers can implement complex decentralized applications and apply blockchain in industries like financial systems, supply chains, and health care [2, 20, 29]. Ethereum represents its state as a collection of key-value pairs in the *Merkle Patricia Trie* (MPT) [16], and thus users can trace the transactions from the *world state*. In Ethereum, transactions are called *smart contract* and run in the *Ethereum Virtual Machine* (EVM) to manipulate the state of the MPT by updating the values of the corresponding keys.

2.2 Throughput Bottleneck

The throughput of blockchain depends on two main factors: block interval and block size. The block interval, which is set by the consensus layer, specifies the frequency at which blocks are generated. For instance, Bitcoin's original Proof-of-Work [25] (PoW) consensus generates blocks slowly, resulting in a throughput of approximately seven transactions per second. However, newer consensus protocols such as Conflux [22] and OHIE [31] aim to increase throughput by generating blocks more quickly, enabling a throughput of more than 5000 TPS, several orders of magnitude faster than the original Nakamoto consensus. Furthermore, Shrec [18] has been developed as a transaction relay protocol to utilize network bandwidth effectively and avoid consensus from being a bottleneck under high transaction throughput scenarios. With these advances, the throughput bottleneck of blockchain systems shifts to the block size.

Researchers have attempted to address the issue of throughput by increasing block sizes. However, this solution has its limits as block size is closely related to the number of transactions per block interval. When the block size increases too much, nodes with lower performance may struggle to keep up, causing poor execution rates. Additionally, larger blocks require more time for propagation and verification by other network nodes, which may result in temporary forks or orphaned blocks.

2.3 Parallel Execution in Blockchain

Most blockchains still execute transactions serially to reach a consensus on the sequence of transactions. However, adding concurrency directly to blockchains can potentially cause contention, thereby wasting the internal parallelism of modern commodity hardware like multi-core processors. Therefore, considerable research attention has been focused on the parallel execution of transactions in recent years. Unlike traditional database management system (DBMS) concurrency control [5], blockchains impose an additional restriction on transaction execution. Transactions must commit in the order specified in the block [23]. Researchers have revised traditional concurrency control algorithms to employ them on blockchains and can be classified into two categories: *optimistic algorithms* and *pessimistic algorithms*.

In optimistic algorithms, transactions are executed in parallel without blocking any operations. However, conflicting transactions may abort or restart once conflicts are detected. Additionally, in blockchain, a transaction can enter the validation phase only if all previous transactions in the block are committed. A typical optimistic algorithm is the OCC, which enables multiple transactions to access and modify data simultaneously. In OCC, each transaction verifies if any other transaction has modified the same data before committing their changes. For example, Garamvölgyi et al. [17] proposed an OCC scheduler with deterministic aborts (OCC-DA) to schedule transactions in public settings.

In pessimistic algorithms, transactions are not allowed to access an object until the corresponding lock is acquired. In blockchain, transactions usually hold all acquired locks until it commits. For example, Dozier et al. [11] used a Pessimistic Concurrency Control (PCC) technique by locking the accounts accessed during transaction execution.

The execution of Ethereum transactions in parallel is challenging because the EVM operations depend on the blockchain's runtime context. For example, to execute an ADD operation, we must know the top two elements in the stack; to execute an MLOAD operation, we must know the memory content at that point[23]. These contexts may differ if we obtain the runtime context without executing all previous operations. To analyze the parallelism of Ethereum workloads, Garamvölgyi et al. [17] performed an empirical study. The study revealed that the majority of data conflicts arise from counters (e.g., balances) and storage. Conflicts on counters commonly result from the concurrent access of two transactions to the same account, while conflicts in storage result from EVM storage operations such as SLOAD and SSTORE. In real-world Ethereum situations, these problems are common as several hotspot contracts emerge, many of which have obvious storage conflicts in related transactions.

3 MOTIVATION

3.1 Parallel Execution in Ethereum

Our research focuses on parallel execution in Ethereum, which presents a more challenging problem than most blockchain systems. This is primarily because Ethereum adopts an account model in its state, and transactions on the same account may cause data races. Additionally, executing smart contracts in Ethereum can modify the world state by updating the values of corresponding keys, causing *data races* in transactions.

Executing smart contract transactions concurrently poses a significant difficulty, as various transactions requested by clients may contain conflicting access to shared metadata. The arbitrary execution of transactions can cause data race issues resulting in an inconsistent world state. According to an empirical study by Garamvölgyi et al. [17], most data conflicts encountered in parallel Ethereum workloads are derived from storage and counters. Conflicts in storage are typically caused by EVM storage operations, whereas the balance exhibits contention when two transactions apply to the same account. In detail, these conflicts can occur due to read-write or write-write contentions and can be divided into three categories: *read-after-write* (RAW), *write-after-write* (WAW), and *write-after-read* (WAR). If multiple transactions within the block access the same key simultaneously, data races can arise in parallel execution.

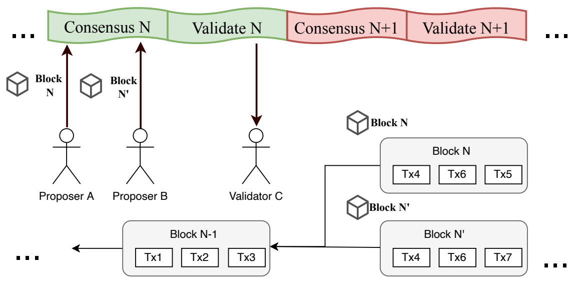


Figure 1: Transaction Execution in Different Contexts: Blockchain, such as Ethereum, adopts a DiCE (Dissemination-Consensus-Execution) distributed computing model. In this model, proposers only produce a single block at a specific height (Block N or Block N') to participate in a new consensus phase. Meanwhile, validators may receive multiple blocks (Block N and Block N') at the same height and have to execute transactions in the block to validate the contents.

3.2 Transaction Execution in Different Contexts

The computing paradigm for blockchains, such as Ethereum, can be described as a *Dissemination-Consensus-Execution* (DiCE) distributed computing model [7]. In this model, transactions are executed in two distinct contexts: proposers are only allowed to participate in a new consensus phase after completing the execution of transactions in the current block, while validators can execute a block of transactions only after the completion of the consensus phase for this block.

Figure 1 further illustrates the two different contexts [10] in transaction execution. The first context is during block formation, where proposers select a batch of requested transactions and execute them in a specific order, recording the final state in a block with the hash of the previous block. Afterward, the proposer proposes it to the network through the consensus protocol. Subsequently, in the second context, validators validate the contents of the block by re-executing the transactions in the proposed block. If the final world state matches the state proposed by the proposer, the block is accepted as valid, and the proposer receives a reward. Otherwise, the block is discarded. As a result, every accepted transaction undergoes execution by all the peers in the system. Besides, transaction execution in validators runs several times more than proposers [7, 10] as validators may re-execute different blocks at the same height due to various fork choice [19] in Byzantium network.

However, the current design of transaction execution is inefficient as it is not performed in parallel and is on the critical path of proposers and validators. Consequently, both proposers and validators execute transactions in a serial manner, which does not utilize the internal parallelism of modern commodity hardware and results in low throughput. This limitation is not unique to Ethereum but is a common feature among many popular blockchains.

3.3 Determinism Levels in Parallel Execution

Previous studies [3, 10, 17, 23, 27] have explored implementing parallel execution on blockchains to increase throughput. However, adding parallelism to transaction execution, especially in smart contract-based transactions, can lead to different executing orders of transactions and cause inconsistency. This introduces a level of non-determinism into the process, affecting the precise timing of node-level transactions[17].

In a blockchain system, achieving consensus among nodes involves determinism to ensure that transactions produce the same outcome on different nodes. Therefore, we have established determinism levels for parallel execution, given the various contexts for proposers and validators. For proposers, transaction execution should be serializable. This ensures that the parallel execution of transactions is equivalent to any serial execution of the same set of transactions. The serial order then becomes the deterministic transaction order, broadcasted to the blockchain network in the block form. For validators, the parallel execution result should produce a specific serial schedule, which refers to the sequence in the block, to ensure the ledger's final state aligns with the block proposed by the proposers and maintain a block consistency. Therefore, transactions should be committed in the same order as their original schedule. Additionally, transactions with WAR, RAW, and WAW conflicts can't be executed concurrently, as they may produce inconsistent results. As an example in Figure 2, four transactions (Tx1-Tx4) are present in the network, and Tx1 and Tx3 access the same key. Proposer A and Proposer B can create different blocks on a blockchain as long as their transactions can be scheduled into a serial order. In this situation, only Tx1 and Tx3 cannot execute in parallel due to updating the same key. If both Tx1 and Tx3 cannot see the update by the other, the transaction that comes later should be aborted. Therefore, Block N from Proposer A and Block N' from Proposer B are both valid. However, if Validator A and Validator B receive Block N from Proposer A, they have to execute transactions according to Proposer A's schedule to produce the same result, meaning their final execution result on different nodes should be equivalent, even though the actual execution schedule might differ.

3.4 Unequal Transaction Quantities

While most blockchains require proposers to process only one block at a given height to receive a reward, the Byzantium network presents a unique challenge for validators. When two proposers produce blocks at roughly the same time, validators may receive multiple blocks at the same height due to the possibility of consensus disagreements and forks. As a result, validators may process multiple blocks, which can be a time-consuming process. A study [6] has shown that validating transactions is a more common activity than mining, and the majority of nodes on the Bitcoin network are primarily involved in transaction validation. In Ethereum, blocks excluded from the main blockchain are called *uncle blocks*. Uncle

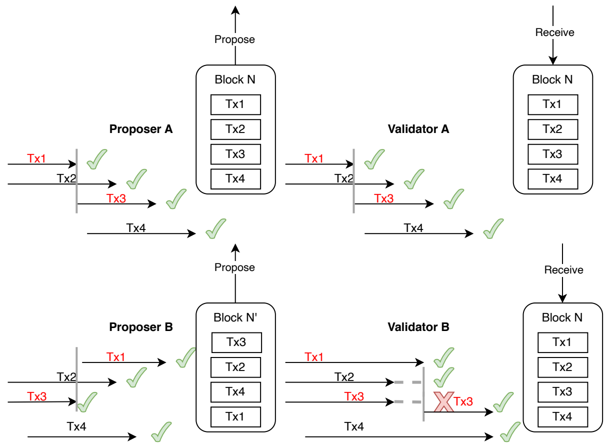


Figure 2: Different Determinism Levels in Parallel Execution: Proposers can produce a block as long as the transactions in the block are serializable, which means schedule #1-#2-#3-#4 in Block N and schedule #3-#2-#4-#1 in Block N' are both valid. However, validators have to commit transactions according to the transaction order in the block which means Validator A and Validator B can only execute transactions in the same schedule as Proposer A when receiving block N (#1-#2-#3-#4).

blocks can also get rewarded as uncle blocks provide a security benefit to the network by reducing the likelihood of a majority attack[9]. Therefore, designing an efficient workflow for validators is crucial for both achieving high throughput and security objectives.

4 SYSTEM DESIGN

This paper presents an overall framework, BlockPilot, for achieving multiple goals in blockchain simultaneously. A prototype system built on Geth is also introduced to demonstrate this framework. Specifically, the framework seeks to achieve the following goals:

- A parallel transaction execution model for proposers and validators.
- An efficient execution workflow in validators supporting processing multiple blocks concurrently.
- Compatible with smart contracts in EVM.

4.1 Overview

Workflow in a Blockchain System. Figure 3 illustrates the overall workflow for executing transactions concurrently and efficiently in a blockchain system by both proposers and validators. Firstly, proposers select transactions from the pending pool and execute them in parallel using an OCC-WSI algorithm described in Section 4.2. These transactions are then packed into blocks and broadcast to the network. Next, validators receive blocks and execute transactions using a pipeline workflow, which can process multiple blocks concurrently, as discussed in Section 4.3. It is important to note that our approach is adaptable to other blockchain systems, although our prototype is only implemented on Ethereum.

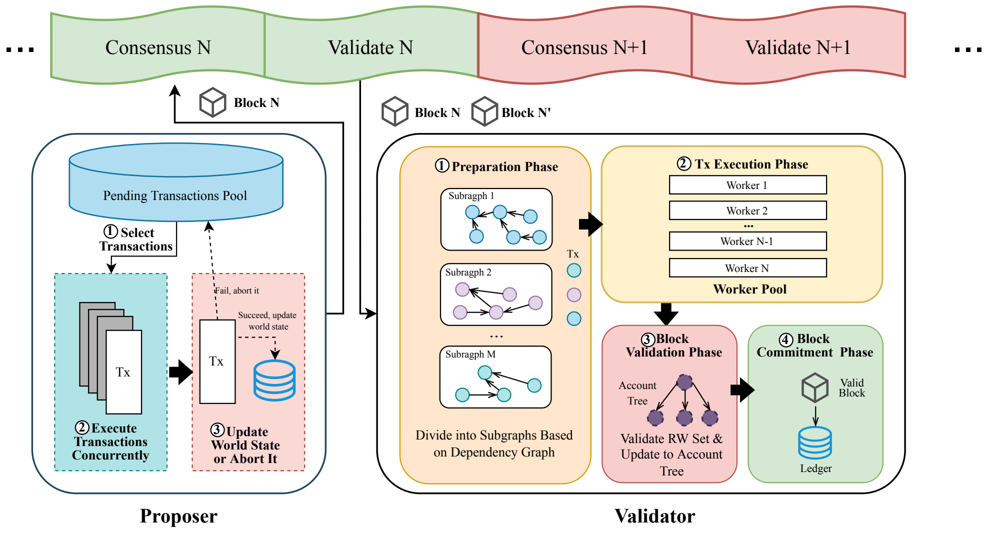


Figure 3: Overall Workflow in BlockPilot: Proposers select transactions from the pending pool and execute them in parallel, utilizing an OCC-WSI algorithm to propose a block to the network. Executed transactions are either packed into a block or aborted back to the pending pool. Afterward, validators receive the block from the network, divide transactions into subgraphs according to the dependency graph, schedule subgraphs into different threads, execute transactions in parallel, and collect the results. In real-world situations, validators may receive multiple blocks, such as block N and block N', at the same height.

Compatibility with EVM. Our design is compatible with the current EVM design, which facilitates portability to Ethereum, one of the most widely used blockchain systems. It supports parallel execution of smart contracts in Ethereum, which can provide benefits in real-world applications.

4.2 Parallel Execution for Proposers

In Ethereum, proposers are responsible for selecting and packing transactions into the next blocks. Typically, transactions with higher gas prices or those from the proposer's accounts are chosen first. Upon selecting the transactions, proposers execute transactions and propose a new block, where they take part in the consensus phase. To execute transactions concurrently with a right result, proposers must ensure that all valid transactions within the produced block can be serializable and scheduled according to their sequence in the block regardless of the conflicts in the shared storage.

Our execution model is based on OCC with *Write-Snapshot Isolation*[4] (WSI) to enforce the serializable property to transactions. The WSI model guarantees that transactions with conflicting reads can only be consistent with the latest version. In contrast, transactions with conflicting writes can be committed to the same block because they can still be scheduled to serial order. After executing transactions based on a snapshot, any transaction with conflicting results is put back into the transaction pool if the result is inconsistent with the latest state.

Algorithm 1 describes the detailed algorithm for our OCC-WSI algorithm. Initially, proposers create a state based on the current ledger. Multiple threads choose transactions from the pending pool based on their preferred strategy. Once a transaction is chosen, it creates a snapshot based on the current state and executes transactions in parallel. After obtaining the execution results, proposers verify whether the results conform to the latest state. Execution results may be inconsistent if intermediate results modify the state and do not include in the current snapshot. If the results are inconsistent with the latest state, the proposer will abort the transaction back to the transaction pool. Otherwise, the proposer writes the results into the block and then updates the state. In distributed consensus, transaction execution results are deterministic, which means the same code triggered with the same inputs will produce the same outputs. Therefore, transaction execution over a specific transaction is executed over a specific storage version on its associated keys. We use a unique version to relate the storage version to the transaction sequence, where each submitted transaction has a snapshot version corresponding to the sequence of the transaction within the block.

In OCC-WSI, executed transactions generate a read-write set (rs&ws) in the execution results, which includes a set of key-value pairs that record read or write operations in <key, version>. Furthermore, a shared table is reserved in <key, version>. Each key corresponds to an account address and is updated by any associated write sets. Initially, each key is assigned with version 0, and each valid transaction will be assigned a unique, increasing version. If multiple threads update the same row at the same time, the *reserve* table is only updated by threads executing compatible transactions and is updated to both the snapshot and table.

Proposers receive rewards only if their blocks are confirmed by the majority of validators in the blockchain. Therefore, it is proposed that they provide execution details like read and write sets about their transactions in the block profile and broadcast it into the network. This enables validators to validate transactions faster which can also benefit the proposers.

Algorithm 1: OCC-WSI

```
Input  : pending transactions $Txs$, thread pool $threads$,
         database state $State(0)$
Output : block $Blk$, block profile $Info$
1  Initialize heap $Heap$ with pending transactions $Txs$;
2  Initialize reserve table $Table$;
3  Initialize block $Blk$;
4  Initialize block profile $Info$;
5  while $curGas < GasLimit$ do
6      for $thread \leftarrow threads$ parallel do
7          $tx \leftarrow$ PopHeap($Heap$);
8          $snapshot(thread, version) \leftarrow State(version)$;
9          $rs, ws \leftarrow$ ExecuteTx($tx, snapshot$);
10         DetectConflit($tx, rs, ws$)
11 return Blk,Info;
12 Function DetectConflit($tx, rs, ws$):
13     for each $rec$ in $rs$ do
14         if Table [rec] > snapshot.version then
15             PushHeap($Heap, tx$);
16             return;
17     $Blk \leftarrow$ AddTx($Blk, tx$);
18     $State(version' + 1) \leftarrow$ ApplyTx($State(version'), ws$);
19     $Info \leftarrow$ AddInfo($Info, tx, rs, ws$);
20     $curGas \leftarrow$ AddGas($curGas, tx$);
21     for each $rec$ in $ws$ do
22         $Table[rec] = version' + 1$
23     Synchronize with all worker threads
```

4.3 Pipeline Design for Validators

After receiving a block from the block network, validators must re-execute transactions within the block to ensure their validity and accuracy in the state transition. Given that the state after executing all the transactions in the block is deterministic, validators must confirm that their committed transactions can be scheduled into the same serial order as that of the proposer and compare the final world state with the proposed block. To correspond to a specific serial schedule in parallel execution, the scheduler schedules transactions according to their read-write and write-write conflicts. The scheduler schedule transactions according to their dependency graph and puts conflicting transactions into the same subgraph. Conflicting transactions are executed in a serial order according to their original order in the received block, while non-conflicting transactions can run in parallel in EVMs. Afterward, all the transactions are committed to the world state strictly according to the block transaction order. For example, Figure 4 demonstrates a dependency graph featuring two subgraphs: Tx2-Tx8 and Tx1-Tx3-Tx4. The scheduler arranges transactions Tx1, Tx3, Tx4 into thread 1 and Tx2, Tx5, Tx6, Tx7, and Tx8 into thread 2. Transactions in the same thread are executed serially, but the two threads run in parallel.

To execute transactions in parallel and process multiple blocks concurrently, we propose a pipeline workflow for validators which comprises four more detailed phases: preparation, transaction execution, block validation, and block commitment.

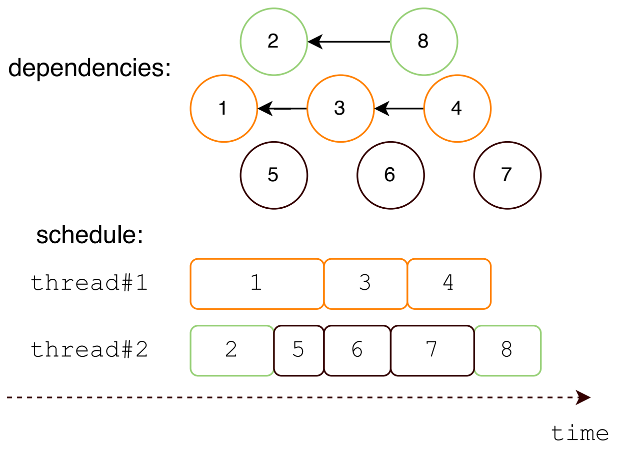


Figure 4: Schedule on Dependency Graph: The scheduler schedules transactions according to their dependency graph. Transactions in the same dependency graph (Tx1-Tx3-Tx4 or Tx2-Tx8) are scheduled into the same thread. Transactions in the same thread are executed serially, but different threads run in parallel.

Preparation Phase. The validator uses a *scheduler* to generate a dependency graph[3] based on conflicts in transactions. The conflicts are detected from the account level because account counters (e.g., balance) are changed in every transaction, and updates to *contract account* can cause the overall update to the account MPT. The scheduler then assigns subgraphs into different threads according to their *gas*[12]. The transaction's gas can serve as a reasonable estimation of execution time because the most time-consuming operations (namely, SLOAD and SSTORE) have very high gas costs [17]. Therefore, the scheduler assigns conflict-free jobs to threads that consume less gas and can have their running time captured properly in most situations.

Tx Execution Phase. As executing transactions is the most time-consuming phase in the pipeline, a *worker pool* is designed to support executing transactions in multiple threads. Specifically, a worker executes scheduled transactions serially in EVM and sends out the execution result to an *applier*, which aggregates transaction information.

Block Validation Phase. The validator utilizes an applier to gather transaction results and verify them. In detail, the applier validates the results of the transactions with the block profile and applies changes to the world state. To be specific, the applier must validate the execution information, such as the read/write set, with the block profile, applying changes to the world state. After receiving all the transactions in the block, the applier then validates the block state with the proposed block.

Block Committment Phase. If the validator verifies that the block is valid, it is ultimately committed to the database.

In the pipeline, validators can process blocks at the same height as well as blocks at different heights. Blocks at the same height can be executed simultaneously in the pipeline, as the world state is not dependent on others. However, blocks at different heights have to wait until all the transactions in the previous block have been received and validated by the applier to ensure they have the right result of the previous block. For example, in Figure 5, block N and block N' can execute simultaneously in the pipeline because they are at the same height N. In this situation, free workers will execute transactions regardless of the block information, so the transaction execution phase may overlap with another block. However, block N'+1 cannot overlap with the previous block N' in the block validation phase because block N'+1 depends on the result of block N' to ensure a block consistent in the blockchain.

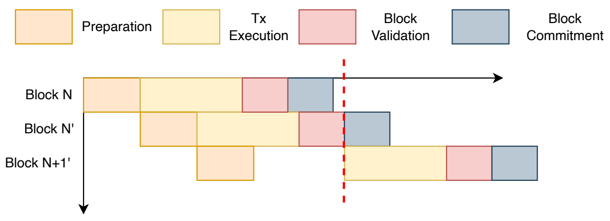


Figure 5: Pipeline Schedule in Validators: Blocks at the same height (Block N and Block N') can be executed simultaneously in the pipeline. In contrast, blocks at different heights (Block N'+1) have to wait until the previous block (Block N') finishes in the block validation phase.

4.4 Assumptions for Block Validation

We assume that honest proposers present valid blocks with accurate block profiles containing the necessary read and write sets. This implies that validators will reject the block if they execute transactions and receive an inconsistent result during block execution. Specifically, Algorithm 4.2 describes the validation process after transaction execution. When transactions in the subgraph complete execution, an applier in the block validation phase will verify the world state and read-write sets. The applier collects read-write sets from workers, checks them against the block profile, and authenticates them. Once all read and write sets in the block profile are verified, the applier confirms the world state aligns with the expected one and moves the valid block to the block commitment phase. Ultimately, the block is then committed to the ledger.

5 EVALUATION

5.1 Experimenetal Setup

We present a comprehensive evaluation of our system on its correctness and performance. Our experiments are conducted on a platform with an Intel i5-13600K processor (14 physical cores), 64GB memory, and a 2TB SSD. The system software is running on Ubuntu 22.04 LTS.

Workloads. To evaluate our system, we use real-world blocks from Ethereum mainnet and compare it with Geth (v1.10.26) as our baseline. The correctness evaluation processes 10 million blocks (from Height 0 to Height 10 million). The performance evaluation processes 100,000 blocks from Height 10 million, where each block contains an average of 132 transactions.

Algorithm 2: ValidateSubgraphs

```
Input  : Block $Blk$, Transactions $Txs$, block profile $Info$,
         result in subgraphs $Results$
//block validation phase
for $res$ in $Results$ do
    if $rs, ws$ in $res$ violate $Info$ then
        Abort Blk;
        return;
$state \leftarrow$ ApplyTxs($Txs, Results$) ;
if $state$ violate $Blk$ then
    Abort Blk;
    return;
//block committment phase
CommitBlock (Blk, Txs);
```

5.2 Correctness Validation

To verify our results, we processed the initial 10 million blocks. In Ethereum, the world state is stored in an MPT structure, and a block includes the hash root of the current world state in the block header. Two world states are considered identical only if their MPT roots are the same. Our prototype system has always produced the same MPT root that matched the values in the block, which demonstrates it follows the rules defined in the Ethereum yellow paper[30] and is compatible with current Ethereum.

5.3 Evaluation of Proposer

To evaluate the performance of proposers, we implemented the OCC-WSI algorithm to process blocks and measured its efficiency. We utilized transactions in real-world blocks as the pending transaction pool for each block, selecting transactions according to the gas and processing a new block as long as all the transactions in the block could be scheduled into a serializable arrangement. To execute transactions in parallel, we utilized 16 threads to execute transactions.

Figure 6 displays the histograms for the distribution of speedup bounds from 2 threads to 16 threads for block exexution in proposers. The vast majority of blocks (99.7%) are successfully accelerated in the model. Proposers obtain an average speedup of 1.82x in 2 threads, 2.60x in 4 threads, 3.56x in 8 threads, and 4.89x in 16 threads. Notably, the speedup rate in proposers shows better performance and scalability than validators because proposers only need to produce a result with a serializable schedule, which makes the parallel execution problem more flexible. However, some blocks cannot achieve a high acceleration because of serious internal conflicts in transactions, explained in Section 5.5.

5.4 Single-Block Evaluation of Validator

To evaluate the validator in processing a single block, we vary worker thread counts from 2 threads to 16 threads while executing transactions. This measures the performance and scalability of the validator in processing a single block. To ensure a fair comparison, we use the prefetching technique, which is implemented in Geth,

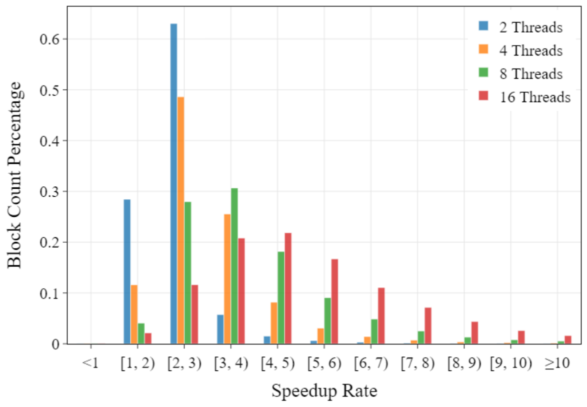
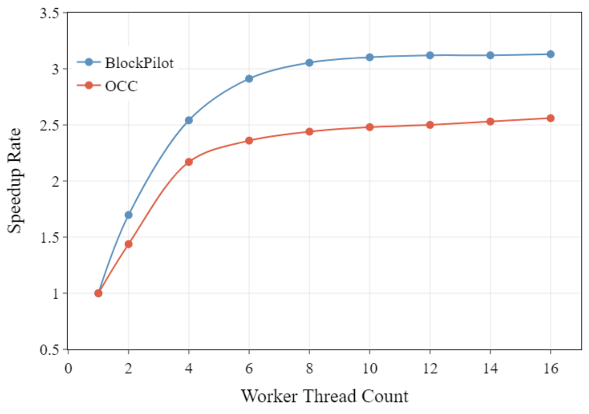
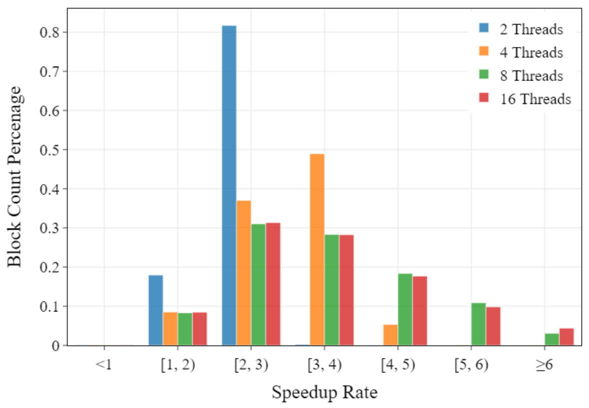


Figure 6: Evaluation of Proposer: Increasing the threads from 2 to 16 results in a steady increase in parallel speedup, indicative of good scalability. Proposers achieve optimal performance in transaction execution, with a 4.89x speedup with 16 threads.



(a) Scalability of a Single Block



(b) Speedup Distribution of a Single Block

Figure 7: Single-Block Evaluation of Validator: Our system accelerates most blocks during transaction execution and scales well up to 6 threads. More threads cannot fully utilize parallelism in transaction execution due to the hotspot problem in Ethereum.

to reduce the I/O impact in executing transactions and prefetch all required storage slots to memory slots.

During the execution of each block, validators generate a dependency graph of transactions by collecting storage read and write traces and assign transactions to different threads to prevent conflicts in execution. Our scheduling approach is based on gas priority, where the subgraph with the heaviest path is selected first to capture the running time.

In Figure 7(a), we illustrate the speedups of BlockPilot and OCC [27] as we increase the number of worker threads. Validators can achieve an average of 1.7x speedup in 2 threads, 2.5x speedup in 4 threads, 3.03x speedup in 8 threads, and 3.18x speedup in 16 threads. Our system scales well to 6 threads, with the speedup rate increasing significantly. However, when the system reaches 6 threads, the speedup rate of the system only increases slightly. Besides, our method is better than OCC method in overall performance.

Besides, as shown in Figure 7(b), our system successfully accelerated 99.8% of the executed blocks, achieving varying speedup rates. Some blocks cannot fully utilize parallelism in transaction execution due to the hotspot problem in Ethereum. Hotspot contracts cause obvious storage conflicts among transactions in a block, reducing the parallelism of blocks over time. Further explanation is provided in Section 5.5. Besides, as we use a gas-based schedule on transactions, it sometimes cannot properly capture the running time, where storage operations are much slower than other transactions [17].

5.5 Effects of Hotspot Problem

According to Saraph et al.[27], the parallelizability of blocks decreases over time due to several hotspot contracts. This problem is even more severe in current application patterns like DeFi, NFT, and token distributions. For instance, Uniswap [1] provides an example of this phenomenon [1], which facilitates Ethereum-based token exchange without intermediaries. However, transactions of similar applications can lead to data races and pose significant challenges to parallelization.

To further explain the hotspot impact on block speedup, we evaluate a relationship between the proportion of the largest subgraph in blocks and the speedup rate when executing transactions with 16 threads. As shown in Figure 8, we observed the largest subgraph contains an average of 27.5% transactions in blocks, which presents a big challenge for parallelism. In most blocks, the longest critical path is more than 20% of transactions in the block, with some even containing more than 40% of transactions in the critical path. As the proportion of the largest subgraph increases, the speedup rate decreases sharply. When the subgraph is small (about 10%), block speedup can achieve more than 4x, while blocks containing only a single subgraph are processed at almost the same speed as the original EVM.

These scenarios can have a considerable impact on parallel acceleration since conflicting transactions can only execute serially, which cannot fully utilize parallelism in real-world situations to accelerate transaction execution. However, contract developers have no incentive to address common storage bottlenecks since the current EVM executes transactions in a serial pattern.

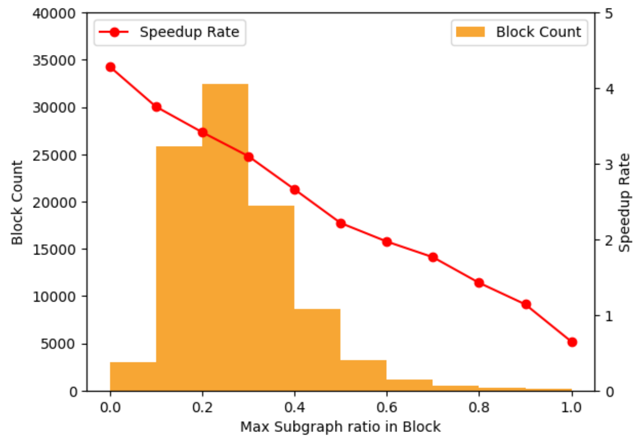


Figure 8: Effects of the Hotspot Problem: As the maximum subgraph ratio increases in blocks, the speedup rate experiences a significant decrease. Furthermore, most subgraphs contain an average of 27.5% transactions in blocks, which indicates that transactions related to hotspots significantly hinder parallel efficiency.

5.6 Multi-Block Evaluation of Validator

To assess the validator's ability to process multiple blocks simultaneously, we simulated executing multiple blocks at the same height by concurrently executing a block multiple times in validators. This evaluation involved executing multiple blocks ranging from 2 to 8, using 16 worker threads, to measure the performance and scalability in handling multiple blocks concurrently. The pipeline design in validators enables it to process multiple blocks concurrently and execute transactions from different blocks in parallel.

The relationship between the number of concurrently processed blocks and the resulting speedup is illustrated in Figure 9. The speedup increases as the number of blocks increases from 1 to 4, reaching a peak of 7.72x. Notably, processing more than 4 concurrent blocks slightly reduces the speedup rate, indicating that the validator can handle a maximum of 4 blocks simultaneously without generating a delay.

To support parallelism, the pipeline workflow in validators introduces additional communication and execution costs, while only 16 threads are available for parallel transaction execution. This results in the drop observed in the speedup for more than four concurrently processed blocks due to the need for workers to shift between different contexts to handle distinct blocks and send out relevant information. Yet, increasing the system's number of workers is likely to maintain the speedup rate and achieve a proportional rise.

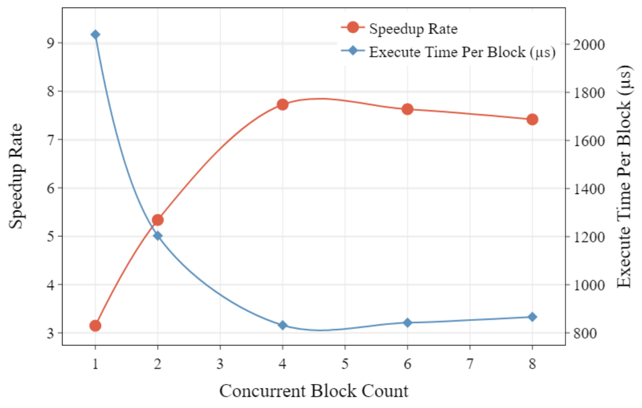


Figure 9: Multi-Block Evaluation of Validator: The speedup rate in validators increases from 1 block to 4 blocks, reaching a maximum of 7.72x. However, the speedup rate decreases slightly from 4 blocks to 8 blocks due to the limited thread number and additional communication expenses.

6 RELATED WORK

In recent years, research has focused on the parallel execution of transactions in blockchain. Saraph et al. [27] conducted an exploratory study to estimate the potential of executing Ethereum transactions in parallel using a 2-phase parallel-then-serial scheduler. They observed the hotspot contract problem on contracts like *CryptoKitties* that cause severe contention and performance degradation. Garamvölgyi et al. [17] identified that data conflicts were caused by counters and defined the levels of determinism in execution. These two works inspire our study, and we reinforce or expand on some of their conclusions. However, these works do not analyze the different contexts in transaction execution and fail to improve the overall performance of transaction execution in blockchain.

Various concurrent control techniques have been proposed to parallelize blockchain transactions. Dickerson et al. [10] described a parallel smart contract execution approach for miners and validators based on techniques adapted from software transactional memory [8]. They use abstract locks to detect conflicts during speculative parallel execution and resolve data conflicts by delaying or rolling back some conflicting invocations. Besides, miners also generate concurrent schedules dynamically so that validators can convert the schedules into deterministic, parallel fork-join programs. Anjana et al. [3] proposed an optimistic software transaction memory (STM) to execute transactions concurrently in miners and produce a dependency graph that validators use to re-execute transactions. On the other hand, Dozier et al. [11] used a Pessimistic Concurrency Control technique by locking the accounts accessed during transaction execution. Similarly, Liu et al. [24] introduced a two-phase lock-commit protocol to support asynchronous execution in multiple groups (or shards). However, most of the proposed techniques are protocol-breaking, as they modify the block structure and the execution semantics, while our approach remains compatible with serial implementations.

Studies have pointed out that the execution of transactions in blockchain systems is influenced by various contexts. In the Byzantium network, validators may have to handle more transactions than proposers due to forks[6]. Kiffer et al. [19] analyzed the large-scale forks in Ethereum and found unique challenges in network partitions. Chen et al. [7] defined Ethereum's decentralized computing paradigm as a *Dissemination-Consensus-Execution* (DiCE) model, where validators and proposers have different contexts in executing transactions. Furthermore, Böhme et al. [6] pointed out that validating transactions is a more common activity than mining in the Bitcoin network, where the majority of nodes are primarily involved in transaction validation. Our discussion of the execution contexts draws inspiration from these works. Given the different execution contexts in blockchain, we defined different levels of determinism in parallel execution for proposers and validators. Besides, we designed a more efficient workflow for validators to execute multiple blocks.

7 CONCLUSION

With the evolution of consensus protocol technology, transaction execution in blockchain has become a bottleneck for execution efficiency, driving the need for parallel execution. In this work, we explain the different contexts in transaction execution in blockchain and define the determinism levels in parallel execution for proposers and validators. We propose an overall proposer-validator parallel execution framework, BlockPilot. Specifically, we design an OCC-WSI algorithm for proposers to support parallel execution, and a pipeline workflow for validators to execute transactions in parallel and process multiple blocks simultaneously. To demonstrate the feasibility of our framework and evaluate its performance, we have implemented a prototype system on top of Geth, the most widely used Ethereum client. Our proposed solution is adaptable to other blockchains that use a Byzantine consensus and have separated roles, such as proposers and validators. As a result, our framework opens the possibility of parallelism of transaction execution in blockchains and enhances blockchain performance.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was supported by National Key R&D Program of China No.2021YFB2700800.

REFERENCES

[1] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. 2021. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.* (2021).

[2] Cornelius C Agbo, Qusay H Mahmoud, and J Mikael Eklund. 2019. Blockchain technology in healthcare: a systematic review. In *Healthcare*, Vol. 7. MDPI, 56.

[3] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 83–92.

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[5] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[6] Rainer Böhme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. 2015. Bitcoin: Economics, technology, and governance. *Journal of economic Perspectives* 29, 2 (2015), 213–238.

[7] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 570–587.

[8] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid Transactional Memory. *SIGARCH Comput. Archit. News* 34, 5 (oct 2006), 336–346. https://doi.org/10.1145/1168919.1168900

[9] Somdip Dey. 2018. Securing majority-attack in blockchain using machine learning and algorithmic game theory: A proof of work. In *2018 10th computer science and electronic engineering (CEEC)*. IEEE, 7–10.

[10] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 303–312.

[11] Ryan Dozier, Sam Ervolino, Zach Newsom, Faye Strawn, and Ross Wagner. [n. d.]. A Correctness Tool to Verify Concurrent Ethereum Transactions. ([n. d.]).

[12] Ethereum. 2023. Ethereum gas. https://ethereum.org/en/developers/docs/gas/.

[13] Ethereum. 2023. Ethereum. Solidity documentation. https://soliditylang.org/.

[14] Ethereum. 2023. Ethreuem. Introduction to smart cntracts. https://ethereum.org/en/developers/docs/smart-contracts/.

[15] Ethereum. 2023. Go Ethereum. https://github.com/ethereum/go-ethereum.

[16] Ethereum. 2023. MERKLE PATRICIA TRIE. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.

[17] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. 2022. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *Proceedings of the 44th International Conference on Software Engineering*. 2315–2326.

[18] Yilin Han, Chenxing Li, Peilun Li, Ming Wu, Dong Zhou, and Fan Long. 2020. Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 238–252.

[19] Lucianna Kiffer, Dave Levin, and Alan Mislove. 2017. Stick a fork in it: Analyzing the Ethereum network partition. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 94–100.

[20] Mahtab Kouhizadeh and Joseph Sarkis. 2018. Blockchain practices, potentials, and perspectives in greening supply chains. *Sustainability* 10, 10 (2018), 3652.

[21] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive block chain protocols. In *Financial Cryptography and Data Security: 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers 19*. Springer, 528–547.

[22] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 515–528.

[23] Haoran Lin, Yajin Zhou, and Lei Wu. 2022. Operation-level Concurrent Transaction Execution for Blockchains. *arXiv preprint arXiv:2211.07911* (2022).

[24] Jian Liu, Peilun Li, Raymond Cheng, N. Asokan, and Dawn Song. 2022. Parallel and Asynchronous Smart Contract Execution. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2022), 1097–1108. https://doi.org/10.1109/TPDS.2021.3095234

[25] Satoshi Nakamoto. 2008. Bitcoin whitepaper. *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)* (2008).

[26] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[27] Vikram Saraph and Maurice Herlihy. 2019. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376* (2019).

[28] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[29] Philip Treleaven, Richard Gendal Brown, and Danny Yang. 2017. Blockchain technology in finance. *Computer* 50, 9 (2017), 14–17.

[30] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[31] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain Scaling Made Simple. In *2020 IEEE Symposium on Security and Privacy (SP)*. 90–105. https://doi.org/10.1109/SP40000.2020.00008

[32] An Zhang and Kunlong Zhang. 2018. Enabling concurrency on smart contracts using multiversion ordering. In *Web and Big Data: Second International Joint Conference, APWeb-WAIM 2018, Macau, China, July 23-25, 2018, Proceedings, Part II 2*. Springer, 425–439.