Utilizing Parallelism in Smart Contracts on Decentralized Blockchains by Taming Application-Inherent Conflicts

Anonymous Author(s)

ABSTRACT

Traditional public blockchain systems typically had very limited transaction throughput due to the bottleneck of the consensus protocol itself. With recent advances in consensus technology, the performance limit has been greatly lifted, typically to thousands of transactions per second. With this, transaction execution has become a new performance bottleneck. Exploiting parallelism in transaction execution is a clear and direct way to address this and further increase transaction throughput. Although some recent literature introduced concurrency control mechanisms to execute smart contract transactions in parallel, the reported speedup that they can achieve is far from ideal. The main reason is that the proposed parallel execution mechanisms cannot effectively deal with the conflicts inherent in many blockchain applications.

In this work, we thoroughly study the historical transaction execution traces in Ethereum. We observe that application-inherent conflicts are the major factors that limit the exploitable parallelism during execution. We propose to use partitioned counters and special commutative instructions to break up the application conflict chains in order to maximize the potential speedup. During our evaluations, these techniques doubled the parallel speedup achievable to an 18x overall speedup compared to serial execution, approaching the optimum. We also propose an OCC scheduler with deterministic aborts, which makes it suitable for practical integration into public blockchain systems.

ACM Reference Format:
Anonymous Author(s). 2021. Utilizing Parallelism in Smart Contracts on Decentralized Blockchains by Taming Application-Inherent Conflicts. In *ICSE 2022: The 44th International Conference on Software Engineering, May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The technical challenge of scaling permissionless blockchains has been a hot research topic for the last few years. With various scaling solutions, be it Ethereum 2.0's sharding or Conflux's Tree-Graph ledger structure, the consensus mechanism ceases to be the performance bottleneck. While disk I/O, network bandwidth, and transaction execution are all possible sources of contention, transaction execution is arguably the most challenging one to address.

Distributed ledgers that follow the account model originally introduced by Ethereum are designed to reach consensus on a sequence of transactions, then process them serially. As a result, current protocols and their implementations are unable to make use of multiple threads on multi-core processors during this execution step. Given the dependencies between transactions through their accesses to a shared data structure called the state tree, the first challenge is to understand how much speedup we can potentially achieve by executing them in parallel. Then, the next challenge would be to design a parallel scheduler with sufficient determinism so that nodes can reach consensus.

To understand the degree of parallelism present in existing transaction workloads, this paper empirically studied a period of historical Ethereum transactions. Taking state access traces (perfect information), transaction gas costs, and the degree of parallelism of computing resources (e.g., 32 threads) as inputs, we constructed an optimal schedule for each block, then compared its execution time to that of serial execution. Our major findings include:

(1) The overall speedup achievable is limited at about 4x compared to serial execution. While there are many blocks whose execution scales with the number of threads, a large portion of blocks performs significantly worse. These results are consistent with previous works. [22, 23]
(2) Most blocks are bottlenecked on a single chain of dependent transactions that need to be executed serially and thus dominate the overall execution time.
(3) A manual inspection of the bottleneck transactions shows that most of them conflict on a single counter or array. From the application's perspective, most bottleneck transactions can be classified into one of three categories: token distribution, collectibles, and decentralized finance.

The empirical study results suggest that, instead of optimizing scheduler implementations, our primary focus should be on eliminating these common sources of contention in smart contracts.

In this paper, we propose a number of different approaches for eliminating the aforementioned bottlenecks. One approach is to use *partitioned counters*, similar to the *sloppy counters* used in the Linux kernel, originally introduced by Boyd-Wickizer et al. [5]. In this approach, we maintain several sub-counters, the sum of which constitutes the value of the original counter. Writes are routed to and operate on different sub-counters based on some attribute, e.g., the sender's address. This way, partitioned counters reduce the probability that any two writing transactions will conflict.

Another approach to addressing bottlenecks is to bypass avoidable conflicts arising from commutative updates on the virtual machine level. Two transactions that both increment a counter but do not use its original value are semantically commutative. However, under the current Ethereum Virtual Machine semantics such increments are translated into a read (SLOAD) and a write (SSTORE) instruction which will lead to read-write conflicts. We propose a

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

new instruction called CADD (commutative add). Two transactions that only have CADD operations but no other reads and writes on a given state entry are not considered conflicting. Increments are applied during transaction commit serially.

Our evaluations suggest that these approaches can raise the amount of speedup achievable to 18x or more, making it approach the optimal case where all transaction dependencies are ignored.

We also note that the non-determinism that is characteristic of parallel execution might prevent blockchain nodes from reaching consensus. A set of incentives for *good* behavior (i.e., following the protocol) and dis-incentives for *bad* behavior (i.e., attacking or misusing the protocol) is an essential part of permissionless blockchains. Ethereum and similar systems offer no incentive to write smart contracts or pack blocks in a way that improves transaction parallelizability. The number of conflicts and/or transaction aborts is a metric of parallelizability that the incentive layer could use to assign financial rewards and penalties. However, under traditional approaches like Optimistic Concurrency Control (OCC) [16], even if we enforce a deterministic commit order, the actual execution on different nodes might still diverge. This would lead to differences in this metric on different nodes and thus it would prevent nodes from reaching consensus.

To address this issue, we introduce an optimistic scheduler with deterministic transaction aborts. To our knowledge, this algorithm is the first of its kind, mostly because distributed ledgers have more stringent determinism requirements than most other domains. Based on our evaluation, this approach allows us to introduce incentives for parallelizability in exchange for a performance impact that is, on average, acceptable.

In summary, this paper's major contributions are recognizing that certain common application-inherent transaction conflicts lead to bottlenecks under parallel execution, providing a set of effective techniques to deal with these, and offering a deterministic scheduling algorithm that makes it possible to incentivize better parallelism.

2 BACKGROUND AND MOTIVATION

Bitcoin [19] introduced *blockchains* with the goal of supporting *cryptocurrency* payment transactions without relying on any central authority. Such a public blockchain is a distributed ledger maintained by a peer-to-peer network in a *trustless* and *permissionless* way. The core piece of this technology is its *consensus protocol*, *Nakamoto consensus*, that probabilistically guarantees the irreversibility of transactions in decentralized public settings, even under adversarial conditions. The ledger is composed of a chain of *blocks*, each of which contains a sequence of transactions, and replicated among all the participant nodes. Each block is generated by a *miner* through some *Proof-of-Work* mechanism, chained at the tail of the valid chain in the miner's view, and broadcast to all the other *validator* nodes through a peer-to-peer gossip network. Due to the latency of block propagation in the network, multiple miners may generate blocks concurrently without seeing the others, and hence may introduce *forks* into the ledger. The Nakamoto consensus employs the *longest chain rule* to let all the honest nodes agree on the valid chain and execute the transactions according to the order of the blocks in the chain and the order of the transactions in each block. The miner of each block on the valid chain gets a certain amount of bitcoin as a reward from the system. The security guarantee is achieved when forks are rare and the ledger basically forms a single chain. In order to avoid forks, the Bitcoin protocol dictates a very low block generation rate in the entire network, which seriously limits its throughput. Specifically, Bitcoin can only achieve a throughput of 7 transactions per second (tps).

Ethereum extends Bitcoin with support for a Turing-complete programming framework, and the Solidity programming language, which allows developers to implement complex *decentralized applications*. This makes it possible to apply blockchain into industries like financial systems, supply chains, and health care [7, 8, 14]. In Ethereum, the *state* resulting from transaction execution is maintained in the form of a *Merkle tree*. Ethereum adopts an *account* model in its state. There are two types of accounts: *user* accounts and *smart contract* accounts. A user account is associated with its *ether* balance information while each smart contract account further has an associated executable code and its own storage represented as a collection of key-value pairs maintained in the Merkle tree. Each transaction occurs between a *sender* account and a *recipient* account. The majority of transactions are one of two kinds: either a *value transfer*, which is a purely monetary transfer of *ether* from sender to recipient, or a *contract call*, where the sender account triggers execution of the code associated with the recipient account. During its execution, a contract call transaction can call functions of other smart contracts. To ensure that transaction execution terminates, each computational step incurs a cost denominated in *gas*, paid by the transaction sender. The sender specifies a maximum amount of gas it is willing to pay (*gas limit*), and if the charge exceeds this value, the computation is terminated and rolled back, and the sender's gas is not refunded. The smart contract code consists of a sequence of bytecode instructions that can be interpreted and executed by the *Ethereum Virtual Machine* (EVM) to manipulate the state of the Merkle tree by updating the values of the corresponding keys. Every bytecode instruction consumes a certain amount of gas. Smart contracts developed using Solidity are compiled into such bytecode sequence before they are published into the blockchain.

Like Bitcoin, Ethereum also employs Nakamoto consensus, although with some different system parameters, e.g., block size, block generation rate, etc. It improves the transaction throughput to about 30 tps but the consensus still remains the major performance bottleneck. In this situation, it makes sense that the EVM is designed as a single-thread engine without the need to introduce parallelism into the transaction execution.

To overcome the throughput bottleneck of Nakamoto consensus, many new and more advanced consensus protocols have been proposed in recent years [3, 11, 12, 17, 18, 21, 25, 26, 30]. These protocols explore alternative structures to organize blocks, e.g., DAG-like structure, together with some novel deterministic block ordering schemes to allow faster global block generation rate without compromising the decentralization and security of the network, and hence the consensus mechanism ceases to be the system bottleneck. For example, both Conflux [18] and OHIE [30] are able to process simple payment transactions with a throughput of more than 5000 tps, several orders of magnitudes faster than the original Nakamoto consensus. Further research work like Shrec [13] also studies and develops a new transaction relay protocol that can more effectively utilize the network bandwidth to prevent it from becoming the new system bottleneck under high transaction throughput scenarios. These techniques shift the throughput bottleneck of blockchain systems to the transaction and smart contract execution, therefore, introduce the emergent need for new technologies that can exploit the parallelism and increase the efficiency of transaction execution.

Some recent research works [2, 4, 9, 10, 20, 22, 23, 31] have explored the designs of a parallel smart contract virtual machine by integrating variant mechanisms of concurrency control. However, according to the reported results, the speed-up that can be achieved by these proposed solutions is far from linear when applied to the real Ethereum workload. We observed that this is mainly because of the lack of inherent parallelism in the real-world workload itself. For example, by investigating the historical Ethereum workload, we found that many critical paths of a series of transactions that have to be executed sequentially are caused by the use of shared global counters. We believe that the essential way to further improve significantly the inherent parallelism of the real workload is to introduce a better programming paradigm that can allow the developers to express parallelism more easily while keeping the original semantics. In addition, in the decentralized environment, driving users to adopt a new paradigm is not that straightforward, as it may incur extra costs, from either the engineering or the economics considerations. Therefore, some new design of incentive mechanisms is required to make the paradigm applicable to real-world applications.

3 EMPIRICAL STUDY

What speedup should we expect when we execute blockchain transactions in parallel? To answer this question, we designed an empirical study using a dataset of historical Ethereum transactions.

3.1 Methodology

We empirically studied the amount of parallelism present in a real-world dataset using historical Ethereum transactions. To this day, Ethereum remains the backbone of the decentralized application ecosystem. As such, this workload represents the most common smart contract interaction scenarios, and the findings can be generalized to many other systems. Our experiment mainly focuses on the period between Jan-01-2018 and May-28-2018 (858, 236 blocks in total), see Sections 3.3 and 7 for a more detailed justification of the dataset used.

The subject of this experiment is smart contract storage conflicts, i.e., cases where two transactions within the same block access the same entry in the state tree, and at least one of these accesses is a write. To obtain these results, we ran an OpenEthereum node (formerly Parity) modified so that it tracks and stores all contract storage accesses. We stored these traces for blocks #1 to #5692235 in a local database. In this experiment, other kinds of conflicting accesses (e.g., conflicts on the account balance) are not considered.

Given that the execution time of transactions is unknown and might vary from node to node, we used the transaction gas cost, obtained from the transaction receipt, as an approximation of this. This follows the practice of a number of related works [22, 23].

Given the transaction dependencies derived from their state access traces and the transactions' gas costs, we constructed a dependency graph for each block. Then, simulating non-preemptive execution on 2, 4, 8, 16, and 32 threads, we constructed an optimal schedule for each block, i.e., a schedule that ensures that no transaction needs to abort while also maximizing thread utilization. Under this execution model, the overall execution cost of this schedule puts an upper bound on the potential speedup that we can achieve; any other schedule might either need to abort and re-execute conflicting transactions, or delay execution through locking. Apart from the overall execution cost (as approximated through the overall gas cost), we also inspected the heaviest path in the transaction dependency graph.

3.2 Results and Findings

Execution Bottlenecks. The experiment shed some light on the limits of speedup we can expect to achieve when executing Ethereum transactions in parallel. We found that the overall speedup on the observed period was only 4x compared to the serial execution, an underwhelming result considering that we had 8, 16, or even more threads available. A closer look at the per-block results shows that in fact, many blocks have much higher speedups, but a significant portion of blocks perform poorly (see Figure 1).

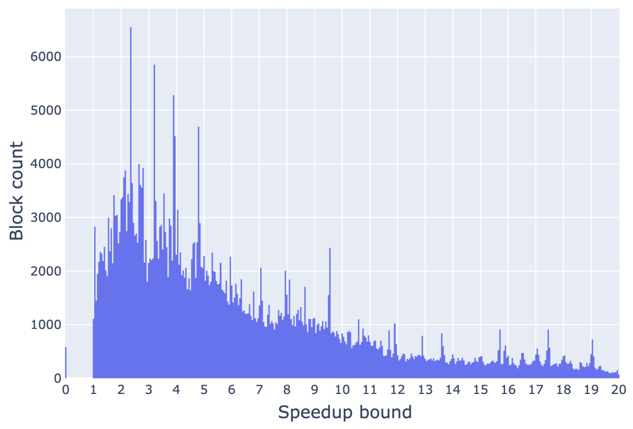

Figure 1: Distribution of parallel speedup bounds

When comparing the execution cost of a block to the execution cost of the heaviest path in its dependency graph, we found that these two often coincide. This means that the overall execution is bottlenecked on the execution of the heaviest path. When we look at single blocks, this heaviest path is often just a single transaction: When, for example, a block has many simple payment transactions and one expensive smart contract call that executes hundreds of token transfers, then this latter transaction will dominate the execution time.

Under our non-preemptive scheduler model and the inherently serial execution model of the EVM, there is no easy way to handle such single-transaction bottlenecks. Our focus, instead, is finding effective ways to handle bottleneck chains of two or more transactions. To focus on these, we re-ran our experiment with batches of consecutive blocks as the unit of execution, instead of just a single block. The idea is that, given thousands of transactions, the relative weight of a single transaction will be much smaller. The same experiment, executed on batches of 30 blocks, shows an overall speedup of 9.46x compared to serial execution. In this case, we observed the same result: Batches are often bottlenecked on a single chain of tens or sometimes hundreds of dependent transactions.

We further examined the impact of these bottleneck transaction chains by re-running the experiment, while ignoring conflicts arising from these smart contracts. The result is an overall speedup of 23.8x compared to serial execution. These results show that bottleneck transactions not only have a crucial impact on the parallelism of our dataset, but also that by breaking up these dependency chains, we can potentially achieve significantly higher speedups.

Classification of Smart Contract Conflicts. To gain a better understanding of smart contract bottlenecks, we collected the primary bottleneck transaction chains for each 30-block batch, and collected the batches that have a speedup bound of 10x or less (3242 in total). Then, we selected a random sample of 200 batches and analyzed them manually. Table 1 shows selected examples from this sample.

In terms of application types, we identified three broad categories: *ERC20 tokens* (token distribution, airdrops) accounted for 60% of the bottlenecks in our sample, *Decentralized Finance* (DeFi) applications made up 29%, while *games and collectibles* (NFTs) were the cause in 10% of the cases.

In most cases, ERC20 tokens lead to conflicts when there are several token transfers over multiple transactions that distribute tokens from the same sender address. Transactions might also have other dependencies, for instance, the total supply is updated every time new tokens are minted. While ERC20 token distributions are heterogeneous in their implementation (e.g. they use various interfaces like `transfer`, `multiTransfer`, `batchTransfer`, `multisend`, `aidrop`), they result in similar conflict patterns.

In DeFi applications like IDEX and Bancor, a common source of conflict is the fee account whose token balance gets updated for every trade. In the case of IDEX, the majority of trades involve ETH, so they all increment the ETH balance of the IDEX fee account.

Examples for games and collectibles (NFTs) include CryptoKitties, Etheremon, and IdleEth. These often involve some globally shared counters, like the number of kitties in the case of CryptoKitties. Maintaining an array of game items is also common. When a game involves payments and rewards, the fee recipient and reward sender account's balance might also lead to storage conflicts.

In terms of the source of conflicts, we found that in 194 of 200 batches (97%) the root cause is one or more counters that get incremented (or decremented) by different transactions. In our sample, the other common source of conflicts, arrays, only accounted for about 2% of the cases.

Bottleneck Code Examples. As an example for counter conflicts in token distributions, let us discuss the example in Listing 1. When calling `transfer`, the sender's balance (`balances[msg.sender]`) is debited, while the recipient's balance is credited. The sender's balance corresponds to one specific storage location in the state tree. The debit operation will compile to a load (`SLOAD`), an add (`ADD`), and a store (`SSTORE`) operation, among others. When two transactions trigger this function from the same sender address concurrently, this will result in a conflict.

```
function transfer(address _to, uint256 _val) /* ... */ {
    balances[msg.sender] -= _val; // <<<
    balances[_to] += _val;
    // ...
}
```

Listing 1: Solidity counters (source: ConsenSys)

Let us look at another example, this time for arrays and collectibles (Listing 2). In the popular CryptoKitties Ethereum game, each new collectible is stored in an array. The `push` operation on Solidity arrays will modify two storage entries: First, it will store the new item at a location derived from the array's length, and second, it will increment its length. Two concurrent transactions will both modify the array length and as such, they will conflict.

```
function _createKitty(/* ... */) /* ... */ {
  uint256 newKittenId = kitties.push(_kitty) - 1; // <<<
  // ...
}
```

Listing 2: Solidity arrays (source: CryptoKitties)

3.3 Generalizability of the Observations

While our observations are based on a relatively narrow period of Ethereum history, it is worth noting that the conclusions are unlikely to have lost their validity, primarily because there has not been any incentive to address this issue. For instance, Uniswap is one of the most commonly used DeFi products as of 2020 and 2021. If we take a look at their `UniswapV2Pair` contract, we can see that the variables tracking token reserves (`reserve0`, `reserve1`) are counters that are updated for every single token swap. The number of transaction storage conflicts and potential bottlenecks are likely to have increased, rather than decrease.

4 AVOIDING APPLICATION INHERENT CONFLICTS

As we have seen in Section 3, a large portion of storage conflicts is associated with storage slots that belong to either counters or arrays. By *counter* here we mean a variable that one can use to track a quantity by incrementing or decrementing it, regardless of its current value. *Arrays* in Solidity are a simple data structure that stores a sequence of elements, along with the number of elements.

In theory, a transaction dependency chain could involve multiple conflicting storage slots. For instance, the chain `#a <- #b <- #c` could mean that `#a` and `#b` conflict on a counter, while `#b` and `#c` conflict on an unrelated array. In practice, however, this is rarely the case. Most transactions in a conflict chain will execute similar operations and will conflict on the same storage entry or entries. In this case, dependencies are *transitive*, i.e., `#c` will conflict with `#a`.

To alleviate the impact of these transaction bottleneck chains, we need to *break them up* into multiple shorter chains by eliminating dependencies between subsets of the transactions involved (see Figure 2). We propose three techniques to achieve this.

Technique 1: Conflict-Aware Token Distribution. In our evaluations, we saw that token distributions (token sales, airdrops) are by far the most common sources of bottleneck conflicts. In the majority of cases, the source of conflict is the storage entry that stores the sender account's current balance.

Table 1: Examples for bottleneck root causes from our 200-batch random sample

block batch	contract	contract type	method(s)	conflict type	conflict source
5536219-5536248	Storj	ERC20	`transfer` (STORJ)	counter	same sender account
5559949-5559978	Free BOB Tokens	ERC20	`airdrop` (BOBx)	counter	`totalSupply`
5497669-5497698	IDEX	DeFi	`trade`, `adminWithdraw`	counter	ETH fee account balance
5493409-5493438	Bancor	DeFi	`quickConvert`	counter	Bancor (BNT) fee recipient
5562289-5562318	CryptoKitties: Core	games/NFT	`breedWithAuto`	counter	`pregnantKitties++`
5562409-5562438	Mythereum Card	games/NFT	`mintSpecificCards`	array	`cards.push(card)`

The simplest way to address these common bottlenecks is to use multiple sender addresses. By distributing the initial funds (where applicable) to a set of sender accounts instead of a single account, and using different sender addresses for consecutive transactions, we can divide the set of bottleneck transactions into disjoint sets of conflicting transactions, each less likely to form a bottleneck.

Of course, the feasibility of this approach depends on the specific implementation of the token. Some tokens have other dependencies: for instance, the total supply of tokens might also be incremented each time new tokens are minted. In the presence of such dependencies, we need a more sophisticated and general approach.

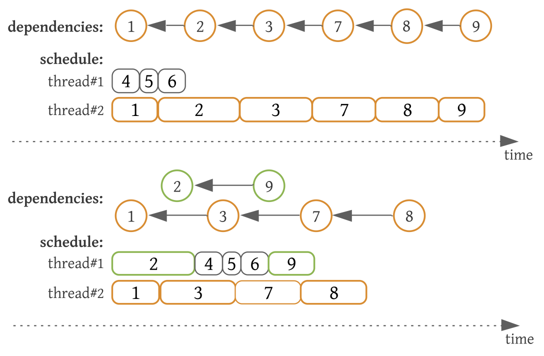

Figure 2: Breaking up a conflict chain into multiple disjunct conflict chains. On the top of the figure, a long conflict chain requires transactions #1-#2-#3-#7-#8-#9 to be scheduled serially on the same thread, dominating the overall execution time. By breaking up this chain into two (#1-#3-#7-#8 and #2-#9), each resulting chain will still need to be executed serially, but the two chains can be executed in parallel to each other. This allows us to achieve a much higher speedup.

Technique 2: Partitioned Counters. Using a technique similar to *sloppy counters* widely used in the Linux kernel [5], we propose a way to route multiple writes on the same counter to multiple distinct storage entries. As writes to different storage entries do not conflict, this technique can drastically reduce the conflict rate.

The main idea of partitioned counters is shown in Listing 3. Here we have a single contract that represents a counter instance. The value of the counter is actually maintained on 10 separate storage entries called *sub-counters*. Each time a transaction modifies the counter's value, we assign a sub-counter based on the transaction's sender address. As addresses are derived using cryptographic hashing, this can be viewed as a pseudorandom sub-counter assignment. When reading the value of the counter, all sub-counters are accessed and their values are summed.

```
contract PartitionedCounter { // LEN = 3
  int256[LEN] public cnt;

  function add(uint32 n) internal {
    uint8 slot = uint8(tx.origin) % LEN;
    cnt[slot] += n;
  }

  function get() internal view returns (int256 sum) {
    for (uint8 i = 0; i < LEN; ++i) { sum += cnt[i]; }
  }
}
```

Listing 3: Partitioned counters implemented in Solidity

Partitioned counters have several advantages. First, a given transaction's writes will all operate on a single storage entry, even if it increments the counter multiple times, as the sender address does not change throughout the transaction's execution. Second, two transactions from two distinct sender addresses that both increment the counter have a much-reduced chance of operating on the same sub-counter and thus conflicts are often avoided. Third, the counter can be adjusted based on the use case, e.g., for counters used frequently one could use more sub-counters, and one could use different criteria for routing transactions to different sub-counters. Our example routes transactions based on the sender address (`tx.origin`) as this addresses common token conflicts.

Partitioned counters have two main drawbacks. First, while we only need to access a single storage entry for writing the counter, reading it will touch all sub-counters. As a result, any transaction that reads the counter will conflict with all writing transactions. As such, this technique is suitable for write-heavy counters. Fortunately, many of the counters we analyzed are never read through transactions. Second, partitioned counters can be significantly more expensive than built-in integers, especially when it comes to reading the counter. This drawback is offset by the potential increase in parallel speedup that partitioned counters offer. Moreover, many counters are rarely or never read in a transaction context.

Technique 3: Commutative EVM Instructions. We have discussed two approaches. One operates on the *application level*, i.e., it addresses conflicts by introducing specific ways to interact with the application. The other operates on the *smart contract level*, by offering tools to contract developers to avoid conflicts. A third approach is to tackle conflicts on the *virtual machine level* by extending the protocol by new instructions that have better conflict tolerance.

When the *Ethereum Virtual Machine (EVM)* executes an increment operation, it first loads the storage entry's current value into memory (`SLOAD`), then modifies this value (`ADD`), and finally it stores the end result back into the storage entry (`SSTORE`). This behavior originates from the Solidity compiler. As discussed before, two transactions incrementing the same counter will both read and write the same storage entry, and so they will conflict.

What is special about incrementing counters is that their current value is only used for calculating their new value, and otherwise it is irrelevant. Put in another way, unlike other read-write conflicts, increments are *commutative*. Two transactions that increment the same counter, but do not use its value otherwise, could be executed in any order. However, under the current semantics of the EVM, such transactions will always conflict.

We introduce special semantics for executing increments in a way that does not result in conflicts. Instead of compiling increments into `SSLOAD` and `SSTORE` instructions, they would instead get compiled into a single `CADD` instruction that stands for *commutative add*. This instruction would take a storage location and a value as its parameters. When the VM encounters a `CADD` instruction, it does not eagerly execute the addition, but instead, it notes this operation down in an in-memory temporary storage. If, at any point after encountering a `CADD` instruction, the VM encounters an `SSTORE` operation on the same storage location, it then erases the pending `CADD` instructions as they have been overwritten. If, at any point after encountering a `CADD` instruction, the VM encounters an `SLOAD` operation on the same storage location, it then first executes all pending `CADD` operations, then use the resulting value for this `SLOAD`.

After the transaction has been executed, the scheduler proceeds to check for conflicts. Concurrent storage reads and writes to the same storage location constitute conflicts. If, however, two transactions only have `CADD` operations on a storage location, but no other reads, then they are not considered conflicting. In this case, these `CADD` operations are executed serially during the commit phase.

While introducing a `CADD` instruction for signaling commutative operations to the VM arguably increases its complexity, it also allows us to avoid a major class of transaction conflicts that originate from operations on a single counter.

5 OCC WITH DETERMINISTIC ABORTS

5.1 Incentives in Parallel Scheduling

Permissionless blockchains have no central authority that could enforce protocol compliance. Instead, protocol designers introduce *incentives* that encourage desired behavior (creating blocks, avoiding storage bloat) and penalize bad behavior (various attacks). The efficiency of parallel schedulers depends on various factors, some of which are under the users' control. Therefore, parallel transaction execution must also come with a set of incentives that maximize its effectiveness.

A detailed design of such a system of incentives is beyond the scope of this paper. We observe, however, that any incentive system must be able to deal with spam or Denial of Service attacks that target mispriced operations and resources in the system, as has happened several times on Ethereum [6]. Parallel execution based on OCC will inevitably lead to some transaction aborts and re-executions. If there is a way for users to intentionally trigger aborts without any penalty, then that opens up the door to a serious DoS vulnerability of the scheduler. Our goal, then, is to define an execution framework that would allow schedulers to deal with this issue by deterministically pricing transaction re-executions.

5.2 Levels of Determinism

Parallel schedulers introduce a level of non-determinism into the execution, as the precise timing of transactions might differ from node to node. This is in direct conflict with the requirements of the consensus mechanism, which relies on strict determinism for the nodes to converge into a consensus state. In blockchain systems, therefore, parallel schedulers must maintain higher levels of determinism compared to traditional algorithms. We define the following three levels of determinism in optimistic transaction execution.

1. **Classic OCC**: Classic OCC [16] has no determinism guarantees. Generally, transactions start execution on a first-come-first-served basis. Node-local consistency is typically ensured by the property of *serializability*, which dictates that the observable results of the parallel execution are equivalent to those of *some* serial execution. However, execution of the same transaction set on different nodes might correspond to different serial executions and yield diverging results.
2. **OCC with deterministic commit order**: Instead of dictating that the parallel schedule is equivalent to *any* serial schedule, it must correspond to a *specific* serial schedule. This means that the final execution result on different nodes will be equivalent, even though the actual execution might differ. This requirement can be satisfied by committing transactions strictly according to the block transaction order, or by scheduling according to a dependency graph [2].
3. **OCC with deterministic aborts**: While deterministic serialization order guarantees that the observable outputs (the resulting state) are the same across all nodes, the actual execution might still differ: Due to different timing of transactions, a transaction might be committed on one node, and aborted on another. If the protocol relies on this commit/abort decision to penalize aborts and avoid DoS attacks (see Section 5.1), this will lead to diverging states. Thus, the highest level of determinism we aim for is when aborts themselves are deterministic: if a transaction is aborted once on one node, it is aborted exactly once on all the other nodes as well.

OCC with deterministic commit order is a topic with considerable research attention in deterministic database systems [1, 27–29]. On the other hand, the stringent requirements of *OCC with deterministic aborts*, to the best of our knowledge, have not been described elsewhere. While imposing such restrictions on OCC schedulers might certainly have a negative impact on the parallel speedup, we argue it is crucial for implementing parallel schedulers under a distributed consensus setting.

5.3 OCC with Deterministic Aborts

Our execution model is based on OCC with *snapshot isolation.* Transactions are scheduled on a set of threads for execution. Executed transactions are committed according to the block transaction order. At the start of its execution, each transaction receives a *snapshot* corresponding to the version of the storage after some transactions preceding it have been committed. This snapshot does not change during the execution of the transaction. The highest transaction id whose committed writes are part of this snapshot corresponds to the *storage version* of the snapshot, or, equivalently, the storage version of the transaction to-be-scheduled.

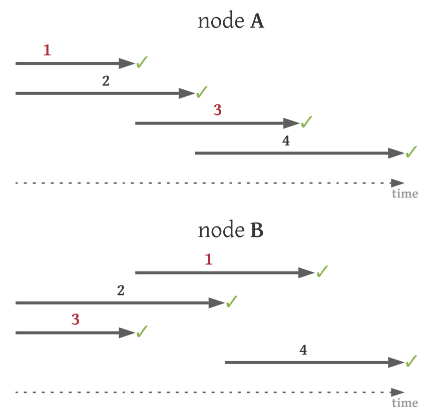

Figure 3: Classic OCC: Transactions are committed right after execution, regardless of their order in the block. This results in different commit orders (#1-#2-#3-#4 and #3-#2-#1-#4) and end states might diverge.

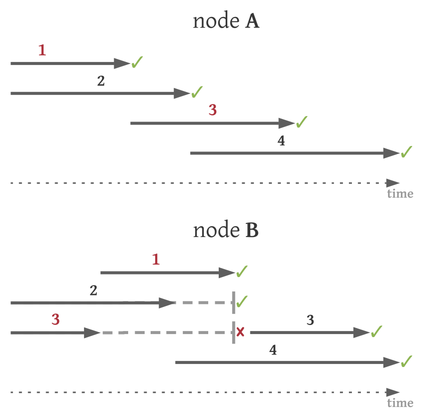

Figure 4: OCC with det. commit order: After execution, commit is delayed until the previous transaction in the block has committed. The commit/abort decision for a transaction might diverge on different nodes (#3).

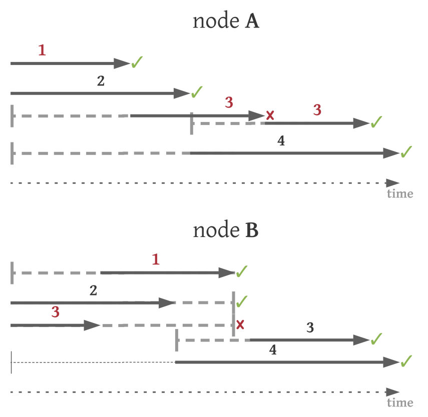

Figure 5: OCC with det. aborts: Transactions can only see a version of the state decided prior to execution, even if a more recent version is available. Each execution of a transaction will either commit or abort on all nodes.

As an example, let us assume that transaction #1 has been committed, transaction #2 is being executed on one thread, and we are scheduling transaction #3 on another thread. In this case, #3 can see storage version #1 (i.e., the contents of storage up to and including #1's writes). If, during the execution of #3, #2 modifies some storage values, these updates are not visible to #3. If, during the commit of #3, the scheduler detects that some values read by #3 were concurrently modified by #2 and thus #3 operated on outdated values, then #3 is aborted and scheduled for re-execution.

In distributed consensus, transaction execution is deterministic: The same code triggered with the same inputs (its parameters and the current state) will produce the same outputs. From this, it is easy to see that a transaction executed over a specific storage version (i.e., the same state) on two different nodes will either commit on both or abort on both.

We then define *OCC with deterministic aborts* as follows. We assign a storage version to each execution of each transaction *prior to execution*: $(tx_n, i) \rightarrow sv_{n,i}$. (tx_n, i) stands for the i's execution of transaction #n, where $i = 0, 1, 2, ...$. Note that, depending on the scheduler implementation, a transaction can be executed two or more times. The last execution must commit, while all preceding executions will be aborted. For all potential executions i of all transactions #n in an execution unit (e.g., in a block), $sv_{n,i}$ is defined uniformly on all nodes, and it is defined prior to execution so that it does not rely on non-deterministic execution details. Then, for any (tx_n, i), transaction #n will either abort or commit on all nodes.

Throughout the execution of (tx_n, i), the scheduler must allow the transaction to access storage entries written by transactions up to and including transaction $sv_{n,i}$. The scheduler must not allow the transaction to access storage entries written by a transaction with an id higher than $sv_{n,i}$, even if it is committed. If $sv_{n,i}$ has not committed and therefore the storage version specified prior to execution is not available when (tx_n, i) is being scheduled for execution, the transaction cannot start execution and must wait.

5.4 Example

We have 4 transactions, labeled #1-#4. Transactions #1 and #3 have a storage conflict: #1 writes a storage entry read by #3. Let us then walk through scheduling these four transactions on two different nodes with 2 threads each, under different determinism guarantees.

Figure 3 depicts an example schedule using *classic OCC*. This approach has no determinism guarantees. In particular, we can see that the commit order on node A is #1-#2-#3-#4, while it is #3-#2-#1-#4 on node B. The diverging relative order of the two conflicting transactions (#1-#3, #3-#1) might lead to diverging states on the two nodes. While #1 and #3 conflict, in this example they are not executed concurrently and therefore neither needs to be aborted.

In Figure 4, we see an example of *OCC with deterministic commit order*. On node B, #3 finishes execution before #1. However, it is is not committed until after #1 has, at which point the conflict is detected and #3 is aborted. The final commit order on both nodes A and B is #1-#2-#3-#4. However, due to the different relative order of the execution of #1 and #3 on the two nodes, the first execution of #3 commits on node A while it aborts on node B. This is a sort of non-determinism is unlikely to be acceptable (see Section 5.2).

Note that #4 on node B cannot read *uncommitted* results from #2, even though these two transactions are executed sequentially. This kind of *snapshot isolation* allows us to avoid *cascading aborts*. An investigation of whether allowing transactions to read uncommitted results is beneficial is beyond the scope of this paper.

Finally, Figure 5 shows how *OCC with deterministic aborts* works. Prior to execution, all nodes decide that the first execution of #3 can only read the state prior to #1's execution ($sv_{3,0} := 0$), while the second execution can see the state after #2 ($sv_{3,1} := 2$). As a result, even though #3 is scheduled after #1 on node A, it is not allowed to see #1's writes and thus it will abort. This yields a result consistent with the other case where #3 is executed concurrently with #1, as on node B. The second execution will see the latest state on both nodes A and B and consequently it will commit on both nodes.

5.5 Assigning Storage Versions

Let us make some remarks about the assignment of storage versions. The simplest approach is to set $sv_{n,0} := -1$. This approach does not rely on any information about the transaction set. While this simple first approximation works, setting $sv_{n,0}$ to -1 (the state prior to transaction #0's execution) will lead to aborts if the transaction set contains any dependencies.

For a more sophisticated heuristic for storage version assignment, we can rely on two kinds of information. First, we can use the expected execution time of transactions to find the latest storage version a transaction is expected to see. If, based on this estimation, #3 will start execution after #1 but before #2, then we set $sv_{3,0} := 1$. Second, an estimation of the transaction dependency graph might allow us to prevent aborts. For instance, if we guess that #3 is likely to conflict with #1, then we can set $sv_{3,0} >= 1$. We do not have perfect information about execution times or transaction dependencies. For the former, the transaction *gas limit* can serve as a reasonable first estimation. For the latter, static analysis and various heuristics might provide us with an approximate dependency graph.

The accuracy of the storage version assignment has a direct effect on the performance of the parallel scheduler: If we use a storage version that is too low, then we risk introducing more aborts. If, on the other hand, we use a storage version that is too high, then the transaction might need to be delayed while it waits for the storage version to become available, leading to thread under-utilization.

Finally, another aspect to consider is the overhead of the scheduler. Maintaining multiple storage versions might introduce a significant storage overhead in case there are many writes. Limiting the lowest storage version each transaction can see might help us put a limit on this overhead.

Algorithm 1: DeterOCC

```
Input: Transactions $T$, gas $Gas$, number of threads $t$, none or a
       dependency graph $D$
1  $H_{txs} \leftarrow$ an empty minheap of $(sv, id)$ ;
2  for $id \in [0, |T|)$ do
3      if $D$ exists then
4          $id_{max} \leftarrow -1$ ;
5          for edge $(id, id_{prev}) \in D$ do
6              //tx_$id$ depends on tx_$id_{prev}$
7              //tx_$id$ reads what tx_$id_{prev}$ writes
8              $id_{max} \leftarrow max(id_{max}, id_{prev})$ ;
9          $H_{txs}.push((id_{max}, id))$ ;
10     else
11         $H_{txs}.push((-1, id))$ ;
12 $H_{ready} \leftarrow$ an empty minheap of $(id, sv)$ ;
13 $H_{threads} \leftarrow$ an empty minheap of $(gas, sv, id)$ ;
14 $H_{commit} \leftarrow$ an empty minheap of $(id, sv)$ ;
15 $next \leftarrow 0$ ;
16 while $next < |T|$ do
17     //Stage 1 : Schedule
18     for $(sv, id) \leftarrow H_{txs}.pop()$ do
19         if $sv > next - 1$ then
20             $H_{txs}.push((sv, id))$ ;
21             break
22         else
23             $H_{ready}.push((id, sv))$ ;
24     while $|H_{threads}| < pool\_size$ and $|H_{ready}| > 0$ do
25         $(id, sv) \leftarrow H_{ready}.pop()$ ;
26         $H_{threads}.push((Gas[id], id, sv))$ ;
27     //Stage 2 : Execution
28     if $|H_{threads}| > 0$ then
29         $(gas, id, sv) \leftarrow H_{threads}.pop()$ ;
30         $H_{commit} \leftarrow (id, sv)$ ;
31         for $i \in [0, |H_{threads}|)$ do
32             $H_{threads}[i].gas \leftarrow H_{threads}[i].gas - gas$ ;
33     //Stage 3 : Commit/Abort
34     while $|H_{commit}| > 0$ do
35         $(id, sv) \leftarrow H_{commit}.pop()$ ;
36         if $id \neq next$ then
37             $H_{commit}.push((id, sv))$ ;
38             break
39         for $id_{prev} \in [sv + 1, id - 1]$ do
40             if $tx\_id_{prev}$'s write set $\cap$ $tx\_id$'s read set $\neq \emptyset$ then
41                 get Aborted ;
42                 break
43         if Aborted then
44             $H_{txs}.push((id - 1, id))$ ;
45         else
46             //Commit successfully
47             $next \leftarrow next + 1$ ;
48 return
```

5.6 The Algorithm

A detailed algorithm for *OCC with deterministic aborts* is presented in Algorithm 1. This algorithm takes a set of transactions and their dependencies as inputs. The dependency graph can be constructed through an estimation of the read-write set of each transaction. It is not necessary for the estimation to be perfect but it needs to be deterministic and consistent on all the blockchain nodes. The more precise it is, the fewer unnecessary aborts we may encounter.

In the beginning, the storage version of each transaction is initialized as the maximum id of the transactions that it depends on according to the dependency graph, or -1 if it has no dependency (lines 2-11). The transactions are pushed into a min-heap H_{txs} indexed by the storage version. There are three other min-heaps. H_{ready} maintains transactions ready to be scheduled, $H_{threads}$ is exactly the thread pool for executing transactions, and H_{commit} is for the transactions that have already finished the execution and wait to be committed. The global variable *next* maintains the id of the next to-be-committed transaction. Note that the algorithm describes the transaction execution mechanism used in our simulation which results in deterministic execution completion order according to the given gas consumption of each transaction. However, in a real system, the correctness and effectiveness of our scheduling strategy do not rely on this execution determinism.

Lines 16-47 show the stages that transactions experience. Stage 1 is scheduling transactions into the thread pool (17-26). We consider a transaction ready to execute when the transaction corresponding to its storage version has been committed. Ready transactions are pushed into the thread pool, if it has empty slots (24-26).

Stage 2 is the execution of transactions in the thread pool. We simply choose the transaction with the minimal remaining gas, which is exactly the top of $H_{threads}$, push it into H_{commit}, and maintain the gas accordingly.

The last stage is trying to commit the transactions one by one in H_{commit} (lines 33-47). Transactions in H_{commit} are maintained in the order of id, since we always commit transactions in order without skips. For each to-be-committed transaction, the algorithm checks whether it should be aborted or committed through checking whether there exist any read-write conflicts between the current transaction and those transactions committed since it starts to execute (lines 39-42). If aborted, the transaction is pushed back into H_{txs} with its new storage version set as $id - 1$, otherwise, the commit succeeds.

6 EVALUATION

6.1 Experimental Setup

The experimental evaluation of the techniques presented in this paper builds on the empirical study discussed previously. All simulations discussed here operate on the storage access traces collected from the Ethereum transaction dataset, as outlined in Section 3.

For evaluating the maximum speedup we can achieve, we rely on perfect knowledge of transaction dependencies. We start by constructing a dependency graph of transactions, where vertices (that correspond to the transactions) are weighted by the transaction gas costs. Then, we simulate scheduling the transactions on a set of threads (2, 4, 8, 16, 32). In each scheduling step, out of all transactions with no unexecuted dependencies, we select the one that has the heaviest path starting from it. The gas cost of this schedule serves as the baseline. For this experiment, we use 10-block batches as the unit of execution, to reduce the effect of single-transaction bottlenecks (see Section 3.2). For evaluating the potential effect of using partitioned counters, we prune the transaction dependencies in a pseudorandom fashion, in a way that is consistent with this technique. For instance, for a counter of length 3, for each dependency, we remove it with a probability of 8/9.

This evaluation assumes that partitioned counters can be applied to all storage conflicts. This is a reasonable approximation based on the results presented in Section 3.2, where we found that almost all conflicts can be traced back to counters used in token distribution scenarios. The information available to our simulated scheduler (EVM bytecode, storage read and write traces) is insufficient to decide whether a storage location corresponds to a counter; for this, one would need to rely on the contract's source code, which is often not publicly available.

For seeing the impact of deterministic scheduling, we implemented an OCC scheduler with deterministic commit order as our baseline. When scheduling a transaction for execution, the scheduler uses the highest committed transaction id as its storage version. To make the transaction commit order deterministic, the scheduler commits transactions according to their block order. For deterministic aborts, instead of using the highest executed transaction id as the transaction's storage version, we use a storage version defined prior to execution: We use -1 (i.e., the storage prior to the block's execution) as the storage version for the transaction's first execution, and use $(txid - 1)$ for its second execution. We then compare their overall gas costs of these two OCC simulations.

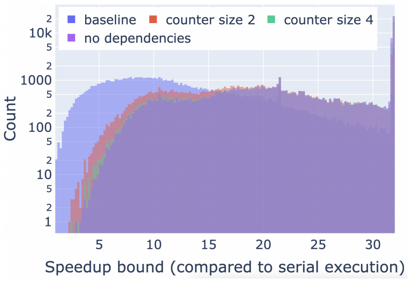

Figure 6: Distribution of speedup bounds (10-block batches)

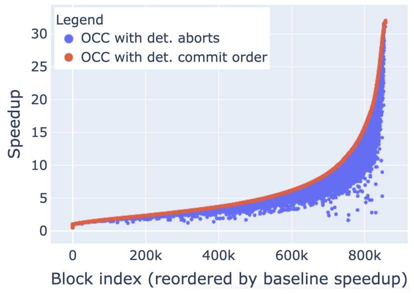

Figure 7: *OCC with det. aborts* performance impact

6.2 Evaluation Results

Overall Results. For each 10-block batch, we look at its *optimal* execution cost on 32 threads (based on the transaction dependency graph), and compare this to its serial execution cost. For the baseline (with no modification), the average speedup over all batches is 11.93x, while the overall speedup on the whole period is 9.25x, due to the bottlenecks discussed in Section 3. Using a counter of length 2, the average speedup becomes 21.23x, while the overall speedup is 17.96x. The highest speedup we can hope to achieve, when we remove all transaction dependencies, is 23.63x on average, while the overall speedup is 20.61x in this experiment.

As for OCC with deterministic aborts, over single blocks with 32 threads, the baseline OCC scheduler has a 3.287x overall speedup (min: 0.52x, max: 32x, avg: 5.89x), while the deterministic scheduler results in 3.275x overall speedup (min: 0.52x, max: 32x, avg: 5.84x). We observed similar results over batches of 30 blocks.

Discussion. These results show that the parallelism inherent in the dataset (9.25x) is much lower than what the transactions would allow for (23.63x). This is due to the fact that transactions depending on each other need to be scheduled serially (or get aborted). By eliminating some dependencies using techniques like partitioned counters, we can approach this limit, achieving up to 17.96x speedup with just a counter size of 2. Figure 6 shows overlayed histograms for the distribution of speedup bounds for each block-batch. From this figure, we can clearly see how partitioned counters let us converge to the optimum, in terms of the parallel speedup achievable.

Figure 7 shows the performance degradation in OCC with det. aborts (blue), compared to OCC with det. commit order (red) on single blocks. For this figure, blocks were ordered by their baseline speedup (red). We can see that extending the scheduler with deterministic aborts did cause performance degradation, however, the speedups generally still do not diverge much from the baseline, except for a few outliers. In fact, in this dataset, 92.47% of the blocks produced exactly the same result using the two schedulers, while only 0.22% resulted in 80% of the baseline speedup or lower.

Implications. These results suggest that partitioned counters can have a significant impact on the highest parallel speedup that we can achieve. Even with just a counter of length 2, when applied to all conflicts, the parallel speedup bound doubled, approaching the optimum. Raising the counter length, we keep approaching the optimum. Based on these results, we believe that the techniques proposed in this paper, when applied to some contracts responsible for some major bottlenecks, can significantly increase the parallel speedup that any real-world parallel scheduler can achieve.

The results about OCC with deterministic aborts suggest that raising the level of determinism only has a minor performance impact, decreasing the overall speedup from 3.287x to 3.275x. As shown in Figure 7, while there are occasional outliers with significant performance degradation under this scheduling mechanism, they are rare. While it is possible that a more performant scheduler, and a workload with more parallelism, will result in a larger discrepancy between these two numbers, based on these initial evaluations, our expectation is that OCC with deterministic aborts is suitable for implementation in real-world blockchain protocols.

7 THREATS TO VALIDITY

The most significant threat to the validity of our study is that transaction and contract interaction patterns have changed since the observed period in 2018 and so our conclusions do not hold for more recent periods. We believe that this is unlikely. The issues pointed out in this paper have not been addressed, and so there has been neither awareness nor incentive to avoid these conflict-inducing patterns. If anything, the problem has likely become more severe, with several new hotspot contracts emerging, many of which have obvious storage conflicts. An example for this is Uniswap, as pointed out in Section 3.3. Saraph et al. [23] also observes that the parallelizability of blocks seems to decrease over time.

There is a chance that the gas cost of transactions does not accurately capture their running time, which would reduce the accuracy of our evaluations. Given that the most time-consuming operations (namely, `SLOAD` and `SSTORE`) have very high gas costs, large deviation seems unlikely.

In this study, we only considered storage conflicts. Other conflict types include conflicts on an account's balance and nonce, and conflicts on contract creation/destruction. Balance conflicts can be handled using partitioned counters. Nonce conflicts require adjusting the nonce management mechanism. As contract existence conflicts are rare, they are unlikely to have distorted our results.

8 RELATED WORK

Parallel execution of blockchain transactions has been the focus of considerable research attention in recent years. Perhaps the first such work is by Sergey et al. [24], in which the authors propose to treat smart contracts as concurrent objects to prevent common bugs. In 2019, Saraph et al. [23] published an exploratory work to estimate the potential benefit of executing Ethereum transactions in parallel by simulating a 2-phase parallel-then-serial optimistic scheduler. They observe a 2-fold speedup for the period in 2018 using 64 threads, and identify *CryptoKitties* as a hotspot contract. They briefly remark on incentives and commutative operations. Reijsbergen et al. [22] evaluate the potential speedup on seven public blockchains using dependency graphs, working on the granularity of contracts instead of storage entries. They report that up to 6x speedup is achievable using 8 or more cores, and observe that larger blocks are easier to parallelize. Our empirical study is inspired by these two works, and we reinforce or expand on some of their conclusions. However, these works do not analyze conflicts in-depth and so they fail to explain the poor parallel speedups they predict. Their models also do not fulfill the determinism requirements that would make them practical in public blockchains.

Numerous previous works have proposed to use various concurrency control techniques to parallelize blockchain transactions. In the approach proposed by Anjana et al. [2], miners use optimistic STM to execute transactions and produce a dependency graph that validators can use to re-execute transactions. Zhang et al. [31], instead of using a dependency graph, propose to include each transaction's write set in the block, and let validators use these to detect conflicts. Pang et al. [20] also consider the granularity of the additional information included in the block. Dickerson et al. [9] propose to use abstract locks to detect conflicts during speculative parallel execution. Dozier et al. [10], on the other hand, use a Pessimistic Concurrency Control technique by locking the accounts accessed during transaction execution. Finally, Bartoletti et al. [4] offer a formal model of concurrent blockchain transactions. Most of the proposed techniques are protocol-breaking, in the sense that they modify the block structure and the execution semantics, while our approach remains compatible with serial implementations. These works show modest speedup on parallel miners but they do not address the root cause of the speedup limit. An overview of this area can be found in the survey piece by Kemmoe et al. [15].

Optimistic Concurrency Control [16] has been widely used in databases and wide-area distributed systems. Deterministic OCC was pioneered by Abadi et al. In Calvin [29], they use a deterministic locking protocol to let nodes arrive on a consistent transaction order, eliminating the need for distributed commit protocols. Their approach is further outlined in several other papers [1, 27, 28]. Our discussion of the determinism of blockchain transaction execution was inspired by these works. In addition to using a predefined serialization order, we introduced an even higher level of determinism, where the effects of transactions that are normally not observable are also deterministic, and can be used for incentive assignment.

9 CONCLUSION

With the evolution of consensus protocol technology in public blockchain, the execution efficiency is becoming the new bottleneck of the entire system, which drives the need of parallelizing the transaction execution. This work observes that the application inherent conflicts are the fundamental obstacle to achieving ideal speedup in existing parallelization techniques. To address this issue, the proposed solution introduces the convenient improvement on the smart contract programming paradigm with consideration of the support of incentives, therefore opens the possibility of maximizing the parallelism of transaction execution in public blockchains.

REFERENCES

[1] Daniel J Abadi and Jose M Faleiro. 2018. An Overview of Deterministic Database Systems. *Commun. ACM* 61, 9 (2018), 78–88.

[2] 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.

[3] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the Blockchain to Approach Physical Limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 585–602. https://doi.org/10.1145/3319535.3363213

[4] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. 2020. A True Concurrent Model of Smart Contracts Executions. In *International Conference on Coordination Languages and Models*. Springer, 243–260.

[5] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Tappan Morris, Nickolai Zeldovich, et al. 2010. An Analysis of Linux Scalability to Many Cores. In *OSDI*, Vol. 10. 86–93.

[6] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. 2017. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *International conference on information security practice and experience*. Springer, 3–24.

[7] Deloitte. 2017. 5 Blockchain Technology Use Cases in Financial Services. http://blog.deloitte.com.ng/5-blockchain-use-cases-in-financial-services/.

[8] Deloitte. 2018. Blockchain: Opportunities for Health Care. https://www2.deloitte.com/us/en/pages/public-sector/articles/blockchain-opportunities-for-health-care.html.

[9] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2019. Adding Concurrency to Smart Contracts. *Distributed Computing* (2019), 1–17.

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

[11] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI*. 45–59.

[12] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 51–68.

[13] 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* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3419111.3421283

[14] IBM. 2020. Blockchain for Supply Chain. https://www.ibm.com/blockchain/supply-chain/.

[15] Victor Youdom Kemmoe, William Stone, Jeehyeong Kim, Daeyoung Kim, and Junggab Son. 2020. Recent Advances in Smart Contracts: A Technical Overview and State of the Art. *IEEE Access* 8 (2020), 117782–117801.

[16] Hsiang-Tsung Kung and John T Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[17] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive Block Chain Protocols. In *International Conference on Financial Cryptography and Data Security*. Springer, 528–547.

[18] 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)*. USENIX Association, 515–528. https://www.usenix.org/conference/atc20/presentation/li-chenxing

[19] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. *Decentralized Business Review* (2008), 21260.

[20] Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2019. Concurrency Protocol Aiming at High Performance of Execution and Replay for Smart Contracts. *arXiv preprint arXiv:1905.07169* (2019).

[21] Rafael Pass and Elaine Shi. 2017. Fruitchains: A Fair Blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 315–324.

[22] Daniël Reijsbergen and Tien Tuan Anh Dinh. 2020. On Exploiting Transaction Concurrency to Speed Up Blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1044–1054.

[23] Vikram Saraph and Maurice Herlihy. 2019. An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts. *arXiv preprint arXiv:1901.01376* (2019).

[24] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *International Conference on Financial Cryptography and Data Security*. Springer, 478–493.

[25] Yonatan Sompolinsky, Shai Wyborski, and Aviv Zohar. 2020. PHANTOM and GHOSTDAG, A Scalable Generalization of Nakamoto Consensus. (2020). https://eprint.iacr.org/2018/104.pdf.

[26] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure High-Rate Transaction Processing in Bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 507–527.

[27] Alexander Thomson and Daniel J Abadi. 2010. The Case for Determinism in Database Systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 70–80.

[28] Alexander Thomson and Daniel J Abadi. 2011. Building Deterministic Transaction Processing Systems without Deterministic Thread Scheduling. In *Proceedings of the 2nd Workshop on Determinism and Correctness in Parallel Programming*, Vol. 5.

[29] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[30] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain Scaling Made Simple. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE.

[31] An Zhang and Kunlong Zhang. 2018. Enabling Concurrency on Smart Contracts Using Multiversion Ordering. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 425–439.