2024

Scalable Transactions in Decentralized Networks

Zachary M. Painter
University Of Central Florida

SCALABLE TRANSACTIONS IN DECENTRALIZED NETWORKS

by

ZACHARY PAINTER
B.S. University of Central Florida, 2018

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida

Summer Term
2024

Major Professor: Damian Dechev

ABSTRACT

The study of shared memory concurrency is extensive. There exist many state-of-the-art strategies for dealing with fundamental concurrency problems, such as race conditions or deadlocks, to leverage massive performance boosts out of modern multiprocessors. With the introduction of blockchain technology as a popular financial tool, we observe many decades-old concurrency problems re-emerge within the context of decentralized networks. These challenges introduce additional constraints, such as the lack of hardware atomic instructions like Compare-And-Swap, or the potential for malicious clients to join the network. In this dissertation, we propose key algorithms which adapt knowledge from the domain of shared memory concurrency to solve emerging concurrency problems in decentralized networks.

We propose three key algorithms which further the state of the art in decentralized networks. (1) We present Hash-Mark-Set, a concurrent algorithm for providing a read-uncommitted view of the blockchain state, enabling a higher success rate in transaction use cases where state changes frequently in relation to the block interval. (2) We propose Proof of Descriptor, a descriptor based consensus mechanism for decentralized networks. Proof of Descriptor utilizes well-known techniques from shared memory concurrent programming to create an efficient and scalable algorithm for blockchain consensus. (3) We propose a descriptor-based algorithm for concurrent execution of smart contracts that efficiently captures the concurrent execution as a graph of descriptors, enabling validators to analyze the concurrent execution and verify its results through re-execution.

I dedicate this dissertation to my parents, Vicki and Steve, and my brother, Joey.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Damian Dechev, for his guidance. He was always readily available to provide detailed comments and insights to myself and fellow lab members. His advice has been a necessary component of my growth as a student, and as a researcher. I am greatly appreciative of his efforts, and for his ability to inspire my best possible work.

I would also like to thank my committee, Dr. Shaojie Zhang, Dr. Paul Gazzillo, Dr. David Mohaisen, and Dr. Eduardo Mucciolo. Each of them has provided high quality suggestions for improving this dissertation, and I greatly appreciate the time they dedicated to listen, read, and discuss my work.

Additionally, I would like to thank my fellow lab members for their collaboration, and friendship. Our discussions over the years have proven invaluable.

Finally, I would like to thank my family, Vicki, Steve, and Joey. Their support and encouragement have emboldened me to be the best person I can be. I would not have been able to complete this work without their support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Moore's Law suggests that the number of transistors within a circuit doubles roughly every two years [27]. Historically, this has resulted in a corresponding doubling of processor throughput for sequential programs. However, recent trends suggest that this single-core speedup is falling behind the curve [41]. The end of Moore's Law, however, does not signal an end to computational scalability. Shared-Memory Multiprocessing (SMP) is a computational model in which multiple processors are linked to a single pool of shared memory. Each processor works independently, and can only communicate with other processors via shared memory. For n processors, this enables at most an n-fold increase in computational throughput. However, this n-fold increase is difficult to achieve in many use cases, due to the difficulty of designing software that can efficiently divide a task between n processors.

Algorithm 1 Concurrency Example

```
1: n = counter++
2: return isPrime(n)
```

Suppose two processors, $p1, p2$, are executing the code sample given by algorithm 1. The intended effect of executing this code with multiple processors would be for each processor to read different incremental values from the counter on line 1, so that they can perform the potentially lengthy $isPrime(n)$ calculation in parallel. However, it is possible for both processors to read the counter at the same time, such that $p1$ and $p2$ observe the same value for n. This negates any potential speedup from multiprocessing, as both processors would perform the same computation, redundantly. This phenomenon is known as a race condition, and is a fundamental problem addressed by the extensive body of literature on concurrent programming.

Although race conditions and related concurrency problems are well understood within the context of share memory multiprocessing, we observe similar problems outside this scope, in the domain

of decentralized networks. A blockchain is a type of decentralized network in which participating clients work to maintain a shared ledger of transactions. A local copy of the ledger is kept by each client, which is propagated to new clients whenever they join the network. In order to achieve a consensus on the state of the ledger, network participants use a consensus mechanism. Consensus mechanisms must enable the network to reach agreement about the state of the ledger despite malicious network participants possibly acting in bad faith. Existing solutions, such as those employed by Bitcoin [28] and Ethereum [4], organize the shared ledger as a sequence of "blocks." Each block contains a sequence of transactions which are executed in order to determine the state of the ledger. In these networks, the consensus mechanism is used to determine which block will be appended to the chain next, creating a sequential chain of blocks.

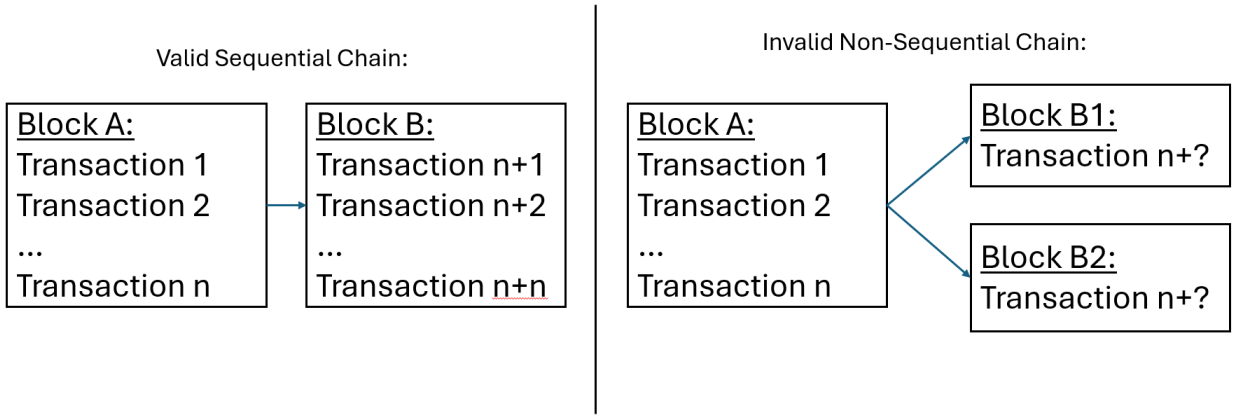

Figure 1.1: The blockchain concurrency problem.

The sequential structure of the blockchain is chosen such that transactions have a total order, that is, each transaction has a definite ordering relative to every other transaction. Although blocks could potentially be appended faster if participants were able to append them in parallel, figure 1.1 demonstrates one of the problems that arises. Assume two blocks, $B1$, and $B2$ are appended to the chain as siblings. In this case, the final state achieved by executing each block in sequence

will differ depending on whether block $B1$ or $B2$ is executed first. Furthermore, it is unclear how future blocks should be appended such that they have a total ordering with $B1$ and $B2$.

We observe that this problem, as well as several others explored in later chapters, is analogous to concurrency problems in SMP. Network participants can be thought of as independent processors, capable of achieving higher throughput so long as they adhere to concurrent principles of correctness. As such, we explore how these SMP solutions can be applied within decentralized networks. In order to apply them, we must address the constraints introduced by the decentralized network. Firstly, networks participants have no source of "shared" memory, nor do they have access to powerful atomic primitives like Compare-And-Swap. Additionally, network participants are anonymous, and may behave maliciously. Any proposed solution must be resilient against a reasonable quantity of bad actors, who are capable of generating an arbitrary number of aliases on the network.

In this dissertation, we propose the following:

- Hash-Mark-Set, a concurrent algorithm for providing a read-uncommitted view of blockchain state. This approach appends a "mark" field to each transaction, which is updated whenever the state changes. This mark can be used to deduce the intra-block state of each object, decreasing the chance of transaction failure due to stale reads. [8], [32]

- Proof of Descriptor, a consensus mechanism for decentralized network which utilizes a descriptor-based algorithm to determine the execution order of transactions within the ledger concurrently, without reliance on a single party to propose a block. [30]

- A concurrent, lock-free algorithm for smart contract execution. Descriptors are utilized during execution to capture all data conflicts between transactions, enabling the concurrent execution to be re-executed deterministically on validator nodes. [31]

CHAPTER 2: BACKGROUND

In this chapter, we overview concepts and approaches from shared memory processing, and decentralized networks.

Concurrent Correctness

For an object in shared memory, it is possible for operations on that object to be invoked by concurrent processes, or threads. In order to define correctness for these types of operations, Herlihy and Wing [19] propose $linearizability$. A concurrent history of operations, H, is linearizable if it can be made equivalent to some valid sequential history of operations without violating any real time orderings within H. Two method invocations, O_1, O_2, have a real time ordering if the response of O_1 precedes the invocation of O_2. Thus, operations that overlap in real time may appear in either order in the equivalent history where each operation was executed sequentially.

Two operations are said to commute if applying them in either order has the same effect. For example, a method $add(x)$ on a set would commute with the operation $add(y)$ if $x \neq y$, but $add(x)$ would not commute with $contains(x)$, because the result of $contains(x)$ is dependent on if $add(x)$ has already occurred.

Algorithm 2 Transaction Example

```
1: if set.contains(x) then
2:     set.add(y)
```

One may want to execute a sequence of linearizable operations uninterrupted, without any concurrent operations taking place between. For example, in algorithm 2, we want to add y to a set only if the set contains x. If x is in the set, but a concurrent operation removed x after the $set.contains(x)$ on line 1, this code would still add y to the set. Such a sequence of atomic operations is referred

to as a transaction [33]. For transactions, Herliy and Wing propose the correctness condition $strict\ serializability$ [19]. A concurrent history of transactions is said to be strictly serializable if the result is equivalent to some sequential history of transactions (linearizability), and operations in differing transactions are not interleaved (commutativity isolation).

Transactional Algorithms

Transactional Boosting [17] is a methodology for executing transactions on a lock-based data structure. Transactional boosting requires underlying linearizable operations to exist on a data structure prior to application. Transactional Boosting assigns an abstract lock [29] to each operation. Method invocations share an abstract lock if they do not commute. To execute a transaction, a thread acquires the abstract lock for each operation, and then executes the underlying linearizable code for the operation. If a thread completes all operations in a transaction, it releases all acquired locks. If a thread fails to acquire a lock at any point during a transaction, it must abort to allow the conflicting transaction to proceed. This ensures non-commuting transactions appear to take place uninterrupted, satisfying the property of commutativity isolation. If a transaction is aborted, any changes written so far must be undone. This is accomplished by executing the inverse operation m' for each operation m executed so far. After all operations are rolled back, the locks acquired by the transaction are released.

Due to the use of locks, this approach cannot guarantee lock-free progress. A thread may be delayed arbitrarily long while holding an important lock. Additionally, the overhead of rollbacks for aborted transactions grows large if transactions conflict frequently.

Lock-Free Transactional Transformation (LFTT) [50], is a descriptor based approach for transactions for node-based data structures. Similar to Transactional Boosting, LFTT requires underlying linearizable operations to exist prior to its application. A descriptor object is an object which con-

tains enough information about an operation that an arbitrary thread would be able to complete the operation. In LFTT, descriptor objects describe each operation in the transaction, and have a field representing the status of the transaction as one of the following: {ACTIVE, COMMITTED, ABORTED}. A descriptor reference is placed in the node class for any LFTT data structure.

In order to execute a transaction, a thread updates the descriptor reference in each modified node to point to the transaction's descriptor, using CAS. If any descriptor reference contains a pointer to a concurrent, active transaction, threads help the competing transaction complete before proceeding with their CAS operation. By updating a node's descriptor reference, an operation is completed "logically," but not physically. This enables aborted transactions to be rolled back cheaply if they abort. A node's logical status can be interpreted by its contents, along with the descriptor it references. If all operations are logically completed, threads execute the underlying linearizable operations, in order. This completes the physical component of the operation, such updating a node's next pointer. Similar to transactional boosting, commutativity isolation is guaranteed because a transaction will only execute after every node's descriptor pointers have been updated to reference the transaction's descriptor. Unlike, transaction boosting, if a CAS-based update fails due to a concurrent transaction, the transaction does not have to abort. Instead, it can retry the CAS-based update after helping the competing transaction complete.

By utilizing CAS instead of locks, LFTT avoids the overhead of locking mechanisms. Additionally, by separating the logical and physical completion of each operation, LFTT avoids the overhead of physical rollbacks whenever a transaction aborts. Due to the thread helping scheme, LFTT experiences significantly fewer spurious aborts than transactional boosting. In LFTT, a transaction only aborts if it is caught in a cyclic dependency with another transaction.

Blockchain Networks

A blockchain is a publicly shared ledger to which network clients may append transactions without reliance on a trusted third party. Blockchains utilize one-way hashing [36] to protect elements of the ledger from tampering. In popular blockchains such as Bitcoin [28] or Ethereum [4], users can submit transactions to the mempool, which are distributed throughout the network in a peer-to-peer fashion. Generally, the network designates a leader via a consensus algorithm to select a block of transactions from the mempool to append to the ledger.

Consensus

Proof of Work (PoW) is a consensus mechanism proposed by Satoshi Nakamoto [28] and is the consensus mechanism of Bitcoin. In PoW, "miners" build blocks of transactions and compete to add them to a shared chain of blocks, which represent the shared ledger. To do so, they seek a "nonce" value which, when hashed with the contents of their block, as well as the hash of the most recent published block, leads with a certain number of zero bits. If a valid hash is found, the miner may append the block to the chain, receiving some compensation for their effort. This approach generated a set of transactions with a total order, transactions are executed in the order given by their blocks, and blocks are executed starting from the first block. It is possible for two miners to simultaneously produce a valid block, producing a fork in the chain of blocks. Clients resolve this by ignoring all blocks that are not part of the longest chain of blocks. Since appending a block to the chain requires a significant expenditure of computational resources, it is not viable for a malicious client to purposefully build their own branch in an effort to undo transactions on the existing chain. To do so, they would need their branch to overtake and exceed the longest branch, which would require the attacker to control more than 51% of the computational power on the network.

Proof of Stake (PoS) is a consensus mechanism in which blocks are chosen by users based on the amount of monetary stake they hold in the network [20]. PoS introduces a class of network clients known as "validators." Validators have a chance to be selected to propose the next block based on the amount of currency they are staking. Validators may lose their stake if they purposefully engage in malicious behavior.

CHAPTER 3: READ-UNCOMMITTED TRANSACTION FOR SMART CONTRACT PERFORMANCE

In this chapter, we present Hash-Mark-Set (HMS). Initially, we discuss the problem of isolation levels for transactions in decentralized networks. We then explain or solution, HMS, which exposes a READ-UNCOMMITTED isolation level for blockchain transactions. Finally, we provide a performance analysis of our solution, demonstrating a significant boost to transaction success.

Motivation

Blockchains rely on a consensus mechanism to agree upon the sequencing of client transactions in a block, committing transactions as a group to the distributed ledger. *Smart contracts* are the interface to process client requests and send transactions to the blockchain peer network. A block of transactions must be validated to ensure that the sequence is consistent. All peers on the network perform the validation step by re-executing the transactions within the block and checking that the initial and final states match, introducing latency.

Latency resulting from the publishing and validation of a block decreases the success rate for the transactions in the block due to the possibility of stale reads of state variables, also known as *storage variables*. Changes to storage variables are only visible after they are committed to a published block. This isolation level of intra-block transactions is called READ-COMMITTED. Transactional reads of storage variables can become outdated while waiting on the validation step since other published blocks may update the storage variables, leading to transaction failure. Additionally, since read operations can only access the published storage variable value, intra-block changes can also cause a transaction to fail due to a stale read.

A smart contract transaction is a concurrent method, often with semantic dependencies. The way block publishing commits multiple smart contract transactions simultaneously is analogous to the way a transactional data structure [17, 50] commits multiple concurrent methods in what appears to be a single atomic step. Using this analogy, the blockchain is a blind transactional data structure that selects and sequences concurrent method calls without regard for their semantics, causing many to fail due to the restrictive READ-COMMITTED isolation level. An ideal algorithm for blockchain transactions would consider transaction semantics and include all related transactions as a series in a block commit.

In this chapter, we detail our solution, *Hash-Mark-Set* (HMS), an algorithm that increases the throughput of smart contract transactions by providing a READ-UNCOMMITTED view of the storage variables. HMS organizes the pool of pending transactions (TxPool) on specific storage variables in a directed acyclic graph (DAG) that establishes an ordering among the transactions and enables an uncommitted view of the storage variables to be retrieved. HMS reduces transactional failures because the READ-UNCOMMITTED view increases the likelihood that a transaction has consistent inputs. Latency is also reduced because concurrent actors will no longer need to wait until a block is committed to see a change in storage variables that is likely to be committed in the next block or two. We integrate HMS into smart contracts through *Runtime Argument Augmentation* (RAA), our proposed technique that allows smart contracts to communicate with external data services prior to sending a transaction.

Challenging Use Cases

Blockchain performance, measured in terms of transaction throughput and latency, is a limiting factor for many use cases [38, 16, 37, 44]. Latency and throughput are considered together in this chapter because the READ-COMMITTED latency of state variable limits the throughput of

successful transactions. This ubiquitous blockchain latency has been dubbed, 'the long system freeze" [14]. Our example use case is a decentralized market to buy and sell assets, a core use case driving blockchain research and investment. This example also represents the general case of concurrent actors reading a time sensitive shared state variable.

Say that trading opens at a certain price, visible to all buyers. Orders are received on the network to be processed. To simplify, orders must be at the exact price, *i.e.* there are no limit or market orders. The price changes frequently and unpredictably due to market dynamics. If 100 orders are received at the published price near the start of a block interval and the price changes after the first order, then only one will be accepted. Blockchain correctness (safety, consistency) is preserved by the expedient of invalidating 99 of the 100 transactions in this example, clearly an inefficient mechanism.

Due to miner privilege, the first order submitted in time may not be the first included in the block. Progress of the system cannot be fair in any case because there is not enough information in the TxPool on which to base a real time order of the requests from different peers. Even with such information, miners are not bound to prevent starvation, quite the contrary they may cause it. Information is also hidden from the buyers querying the smart contract for the price. Block replay is not available within the smart contract. Unless it is separately analyzed, 98 of the 99 price changes are invisible to participants and valuable market information about intermediate price changes is lost. The arbitrary transaction priority combined with read latency also creates a vulnerability known as *blockchain frontrunning* [39].

Methodology

This section presents Hash-Mark-Set (HMS), an algorithm that overcomes the limitations of the READ-COMMITTED isolation level by providing a READ-UNCOMMITTED view of storage variables. The READ-UNCOMMITTED view alleviates the problems demonstrated in the previous section. Clients can observe partial changes within the block prior to publishing, reducing the chance that a transaction will fail due to a stale read. The *Mark* in HMS also establishes a partial intra-block order that a cooperating miner can enforce. Such cooperation is reasonable given financial incentives that might be offered by decentralized asset exchanges.

HMS provides a READ-UNCOMMITTED view by maintaining the transactions in a directed acyclic graph (DAG) that represents an ordering among the transactions in the unprocessed transaction pool, *TxPool*, and applying a topological sort to the longest branch to retrieve the value of an unpublished storage variable. To enable the READ-UNCOMMITTED view to be accessible through smart contracts, we propose Runtime Argument Augmentation (RAA), our proposed technique that modifies the Ethereum Virtual Machine (EVM) interpreter to apply the HMS algorithm and access the value of an unpublished storage variable. The RAA technique is made available to users through our proposed smart contract *Sereth*.

To evaluate the performance benefits of our proposed methodology, we present a new metric, *state throughput*, which measures the throughput of successful transactions. State throughput disregards failed transactions in the throughput measurement, which provides a better representation of the rate at which state changes are made in comparison to raw throughput. In the following subsections, we define state throughput, provide the Sereth smart contract application programming interface, and explain HMS and RAA, the two innovations of this approach.

State Throughput

Blockchains are different from databases in the following way: failed transactions are included in the persistent shared ledger. Because a block may include a large percentage of failed transactions, raw throughput of transactions per second is not an adequate measure of performance. In the example described in Section 3 (*Challenging Use Cases*), raw throughput was 100 per interval, but 99 of 100 transactions fail. In a database these rolled back transactions would not count in throughput, but in a blockchain they are included in the block. A new metric, *state throughput*, T_{state}, is defined here as the product of the raw throughput and the ratio of transactions included in a block that successfully make state changes. State throughput divided by raw throughput yields the transaction efficiency η.

$$\frac{T_{state}}{T_{raw}} = \eta \tag{3.1}$$

Transactions in the TxPool form a concurrent history, with a non-deterministic outcome. We observed that transaction failure can be reduced by obtaining a view of state that is more likely to be consistent at the moment the transaction is committed to a block. To maximize η, transactions are organized to provide a predictive view of state, ordering transactions such that the order closely matches the real time order in which the transactions were received.

Sereth Smart Contract

Our implementation of HMS for Ethereum is called Sereth, a variation of Geth, the name of the standard client. Sereth is implemented as an interoperable Ethereum client that can be substituted for one or more peers in any standard Ethereum network, public or private. The Sereth smart contract shown in Listing 3.1 manages the price and accepts the set and buy transactions from addresses on the blockchain. The $mark$ and get functions are read only. They do not create

Listing 3.1: Sereth smart contract.

```
pragma solidity ^0.4.24;

contract Sereth {
...
// Mark, Set and Get are methods on state variables
// managed by the Hash-Mark-Set algorithm.

    function mark(bytes32[3] raa)
            private pure returns(bytes32) {
        return raa[1];
    }

    function set(bytes32[3] fpv) public {
        // If mark is valid, set new mark and value.
        if (keccak256(fpv[1]) == keccak256(p[1])) {
            nSet++;
            p[0] = bytes32(msg.sender);
            p[1] = keccak256(fpv[1], fpv[2]);
            p[2] = fpv[2];
        }
    }

    function get(bytes32[3] raa)
            public pure returns(bytes32) {
        return raa[2];
    }

// Function buy() demonstrates a dynamic pricing use case
// for the Hash-Mark-Set transactional data structure.

    function buy(bytes32[3] offer) public {
        // If mark and price match then buy() succeeds.
        if ((keccak256(offer[1]) == keccak256(p[1])) &&
            (keccak256(offer[2]) == keccak256(p[2]))) {
            nBuy++;
            p[0] = bytes32(msg.sender);
        }
    }
}
```

transactions but are used to return the intra-block state that will be used in set and buy. This intra-block state view uses RAA to get the results of the HMS algorithm. The values are written into the function arguments using RAA and then returned to the calling address.

Hash-Mark-Set

Hash-Mark-Set takes advantage of an underutilized communication channel among the peers on a blockchain, the transaction pool (TxPool). We created a smart contract, `Sereth.sol`, to manage the state variables. In Sereth, function arguments are formatted so they contain three key elements within the transaction, $address$, $mark$, and $value$. The $address$ field contains the address of the sender of the transaction. The $mark$ field contains a Keccak256 hash [34] which solidifies a transactions place in a series of Sereth transactions. The $value$ field indicates how the sender

would like to modify the state variable. Together, these elements are referred to as a transaction's AMV. To create a transaction using the Sereth contract, one must pass in three parameters: $flag$, $previous\_mark$, and $value$. These parameters are referred to as the FPV. The FPV is easily visible as a string of bytes within the transactions $input$ field.

We define a transaction's $mark$ such that given Txn_1 which follows Txn_0,

$$Txn_1.mark = Keccak256(Txn_0.mark, Txn_1.val)$$

This creates a sequentially consistent ordering between any number of transactions in what we call a $series$. To create a $series$, the FPV of each transaction in the TxPool is extracted from their respective $Data$ fields. By matching the $previous\_mark$ of a transaction with the $mark$ of a different transaction, we can determine a strict order of all Sereth transactions in the current TxPool. This provides the smart contract with a ***Read-Uncommitted*** view of the intra-block state. In addition, because every state change is linked by a unique hash that includes the value, multiple state changes sequenced in the atomic block update are preserved.

Algorithm 3 shows the HMS algorithm as implemented on the Ethereum blockchain. Users interact with the algorithm through an Ethereum contract. We refer to line x of algorithm A as A:x.

Algorithm 3 Transaction Serialization Algorithm

```
1: procedure HASHMARKSET(INPUT)          ▷ Serialize a blockchain transaction pool
2:     RAA ← input
3:     txnList ← PROCESS(TxPool)                                  ▷ Filter TxPool
4:     if len(txnList) == 0 then
5:         RAA ← specialValue
6:         return
7:     series ← SERIES(txnList)                                   ▷ Create series
8:     RAA ← COPY(series.tail.FPV)
```

A call to HashMarkSet() is made from the EVM interpreter when the transaction being processed has a function signature that matches that of a Sereth transaction. The RAA variable on line 3:8

Algorithm 4 Process Transactions

```
1: procedure PROCESS(TXPOOL, INPUT)                    ▷ Filter TxPool for HMS transactions
2:     filteredList[]
3:     for txn ∈ TxPool do
4:         if SIGNATURE(txn) == "set" & SUCCESS(txn) then
5:             txn.FPV ← txn.input
6:             txn.mark ←
                       Keccak(txn.FPV[1], txn.FPV[2])
7:             filteredList.push(new Node(txn))
8:     return filteredList
9: procedure SUCCESS(TXN)                    ▷ Determines if a transaction succeeded or not
10:    FPV ← txn.input
11:    if FPV[0] == successFlag —— FPV[0] == headFlag then
12:        return true
13:    return false
```

represents the storage variable value obtained using the RAA technique. We first extract the RAA from the given $input$ field of the transaction we are processing. This process is simple, as each element is stored in a contiguous 32 bytes within $input$. By writing the result of HashMarkSet() to RAA, the result will be made visible within the contract's execution.

Algorithm 4 details how the current transaction pool is filtered and then returned to the main function for handling.

For each transaction in the pool, we check that the function signature is equal to one of the write functions from our HMS contract. Additionally, we check the first 32 bytes of the FPV for a flag indicating one of several possible states for the transaction. Due to this filtering only a small percentage of the TxPool requires processing, so the overhead of HMS is relatively small.

First, the transaction may be one of the first HMS transactions that appeared during the current block. In this case, we consider the transaction a $head\ candidate$, meaning that it or another transaction with the same flag will serve as the head of the serialized list of transactions for the current block. This allows us to easily continue the list from the previous block without being able

to view the state variable. The second possible state indicates that the transaction is not a head candidate, and at the time of the transaction's submission, it was found to be the successor to the current tail of the series. If a transaction contains neither of these flags, it is considered rejected and is not included in the list of relevant transactions. If a transaction is accepted, The FPV is then extracted from the $input$ field. The FPV contains $previous\_mark$ and $value$, which are the two values needed to calculate the $mark$ of a transaction and determine its place in the series. A node is created from the transaction for later inclusion in a linked data structure.

Once $txnList$ has been populated by transactions from the TxPool on line 3:3, we check on line 3:4 if the list is empty. If so, the submitted transaction is the first Sereth transaction sent in the current block, and the way to know if it matches the previous mark is to check the state variable within the contract. In this case, a flag is written to the $data$ field, which will be visible to the contract. The contract value will be written in the last 32 bytes of the transaction FPV by the sender.

If the list contains one or more transactions, then we know that there already exists at least one series for the current block. Algorithm 5 contains the functions which return the most valid series from a list of Sereth transactions.

Line 5:1 refers to Series(), which iterates through each transaction in the list of Sereth transactions and forms graph relations between all transactions with corresponding mark/value hashes. Due to the uncertain nature of concurrency, it is possible for a transaction to have multiple potential successors, but only one predecessor.

At line 5:9 we locate from multiple potential head nodes the one that produces the deepest graph. From that graph, the deepest branch is our series. This logic mirrors that of the blockchain, in which branches are resolved by taking the longest branch.

Algorithm 5 Create a Series

```
1: procedure SERIES(TXNLIST)                ▷ Create a serialized list from a set of transactions
2:     for txn ∈ txnList do
3:         for txn2 ∈ txnList do
4:             if txn.mark == txn2.FPV[1] then
5:                 txn2.prevTxn ← txn
6:                 txn.nextTxns.push(txn2)
7:     highestDepth ← 0
8:     longestSeries ← nil
9:     for txn ∈ txnList such that txn.FPV[0] == headFlag do
10:        depth ← 1
11:        path ← [txn]
12:        maxDepth ← 0
13:        maxPath ← []
14:        DEEPESTBRANCH(txn, depth, &maxDepth,
                                        path, maxPath)
15:        if maxDepth > highestDepth then
16:            highestDepth ← maxDepth
17:            longestSeries ← maxPath
18:    return longestSeries
19: procedure DEEPESTBRANCH(HEAD, DEPTH, PATH, MAXDEPTH, MAXPATH) ▷ Recursively
    find deepest branch
20:    if len(head.nextTxns) == 0 then
21:        if depth > maxDepth then
22:            maxDepth = depth
23:            maxPath = path
24:        return
25:    for txn ∈ head.nextTxns do
26:        path.push(txn)
27:        DEEPESTBRANCH(txn, depth + 1, path,
                                    maxDepth, maxPath)
28:        path.remove(txn)
```

Runtime Argument Augmentation

Blockchain oracles provide a secure and verifiable medium for smart contracts to access external data feeds, but still suffer from stale reads due to latency. In our implementation of HMS it became clear that a traditional oracle would not satisfy the requirement for intra-block data. To overcome

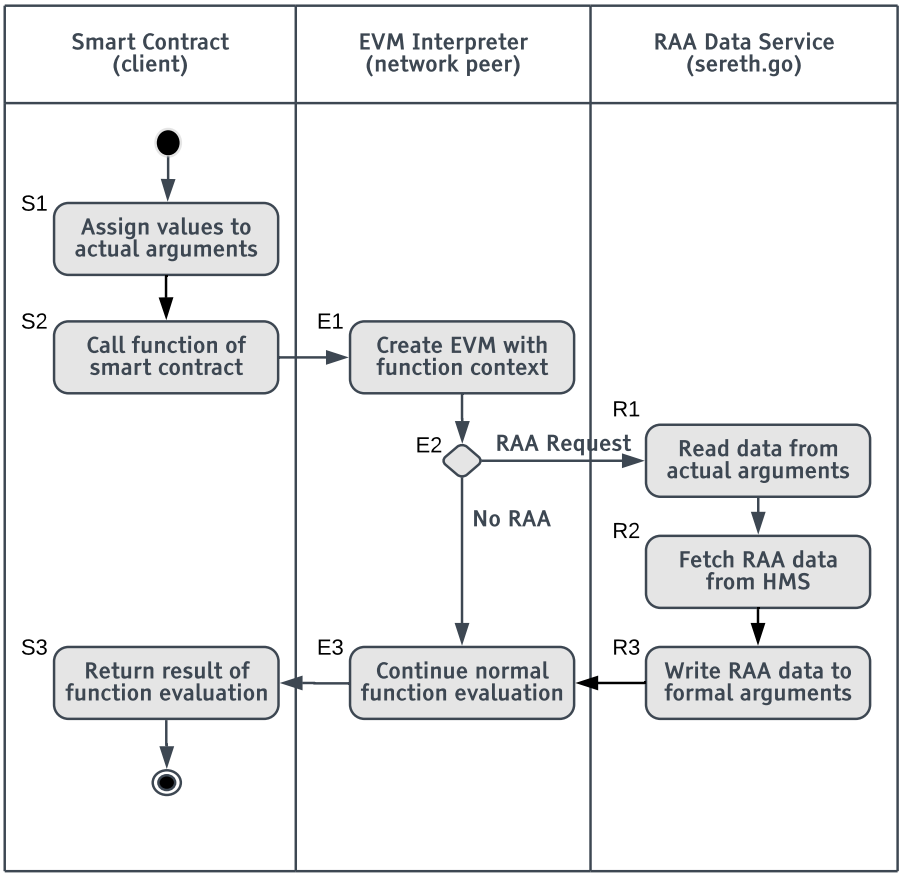

Figure 3.1: RAA activity diagram.

the limitations of oracles, we propose *Runtime Argument Augmentation* (RAA), a technique that provides data to a smart contract by using the argument list as a channel to pass information. RAA is a modification to the Ethereum Virtual Machine (EVM) interpreter, written in Golang. Figure 3.1 is an activity diagram showing the modified processing. In activity $E2$ the EVM interpreter checks to see if a function is requesting external data items using RAA. If so, the interpreter calls the RAA provider in activities $R1$ to $R3$, implemented as a Golang service compiled into the EVM. Data is obtained from the RAA provider and written into the function arguments. The data types of the items being requested must match the data types of the arguments. In $E3$ the function returns

the result of evaluation using the modified arguments to the smart contract for use in activity $S3$. RAA is flexible: any computation can be accomplished by the RAA provider, and the information can flow in both directions. RAA is fast because it is written as an extension of the EVM. A smart contract using RAA is indistinguishable to unmodified clients running Geth, who merely see that arguments are passed in and a result returned.

There are some limitations. RAA cannot be used to modify the arguments of a smart contract function that may send a transaction. This is because transactions along with their inputs are cryptographically signed by the sending address, stored in parameters `msg.hash` and `msg.sender`. Without this protection a malicious Geth client could modify the inputs of a transaction, for example doubling the price offered for an item or changing the delivery address. In testing the limits of RAA we found that the modified transactions would still be mined, but would not be accepted by peers who must validate the newly created block. In order to use RAA information in a transaction, a smart contract or other blockchain actor calls the RAA function first, then uses the information provided to improve the subsequent transaction. This is the process used to obtain the experimental results that follow.

Correctness

Concurrent systems are expected to satisfy correctness (safety) and progress (liveness) properties. Correctness is determined according to a defined correctness condition presented in literature [18]. HMS is designed for the sequential consistency correctness condition because miners are required to preserve the nonce order when committing a transaction from a given thread to a block. Since the nonce is a counter that reflects the sequential ordering of transactions issued by the same thread and a blockchain transaction is analogous to a concurrent method, the blockchain is inherently sequentially consistent. In the following lemma we show that HMS generates a series that provides a

sequentially consistent ordering of the transactions in the longest branch. The benefit of generating a series of transactions in the longest branch is that it offers the greatest potential for optimum state throughput.

Lemma 1. *The series generated from HMS preserves a sequentially consistent ordering of transactions invoked in the longest branch of the directed acyclic graph.*

Proof. For each transaction T in the transaction pool, if the signature is a set operation, and T is either a possible head candidate or is a successor to the current tail in the series, then T's mark is updated by hashing the predecessor transaction's mark and value, and the list of transactions to be considered for the series is amended to include transaction T. If the length of the list of transactions is larger than one, then HMS generates a series of transactions by calling the SERIES function with the transaction list as input. It now must be shown that the generated series is both sequentially consistent and the longest branch. The SERIES function creates an adjacency list of all transactions in the transaction list such that a transaction T_2 that is a member of T_1's list indicates that T_2 is a successor to T_1. The SERIES function then iterates through the potential head candidates and applies the DEEPESTBRANCH algorithm. Each recursive call to DEEPESTBRANCH will iterate through the list of successor transactions in the adjacency list and apply DEEPESTBRANCH to each successor transaction until a transaction with no successors is reached. At each recursive call to DEEPESTBRANCH, transaction txn passed as an input parameter is amended to the path. Since the exploration of the adjacency list guarantees that all successor transactions are visited after a predecessor transaction, any path generated from DEEPESTBRANCH will be sequentially consistent because the program order established in the adjacency list is preserved. Since the depth at each recursive call of DEEPESTBRANCH is incremented by one, and a path that exceeds the maximum depth is recorded upon termination of the recursive calls, the final recorded path by DEEPESTBRANCH will be the longest branch within the adjacency list. □

Progress of the underlying blockchain (the computer) is assumed. We focus here on the progress of smart contract methods using a view of state variables managed by HMS. *Lock freedom* is defined as ensuring that some concurrent actor makes progress, and this is true for the blockchain as a whole but not for an individual smart contract. Miners may assign a low priority to a particular contract so it makes no progress. At peak times, many more transactions are sent to the network than can be included in a block. Transactions sent may be lost due to network failures, memory limitations or peers not replaying them. Miners may refuse to include transactions for arbitrary reasons. As a result, the progress guarantee provided by Ethereum is smart contract termination [1, 24]. Since the TxPool is a finite list of transactions, Algorithm 4 trivially terminates. Algorithm 3 and Algorithm 5 terminate given that the recursive function DEEPESTBRANCH terminates. We now show in the following lemma that DEEPESTBRANCH terminates.

Lemma 2. DEEPESTBRANCH *presented in Algorithm 5 is guaranteed to terminate.*

Proof. The $txnList$ in the SERIES function is a finite list of transactions because it is a subset of the TxPool generated by the PROCESS function. Therefore, each list within the adjacency list of transactions constructed by the nested for-loop on line 2 of Algorithm 5 will also contain a finite number of transactions. Since the $txn.mark$ computed by PROCESS establishes an ordering among the transactions in $txnList$, the adjacency list of transactions will not contain any cycles due to the if-statement on line 4 of Algorithm 5. DEEPESTBRANCH will be invoked by the SERIES function no more than the number of transactions contained in $txnList$. For each invocation of DEEPESTBRANCH, a recursive call to DEEPESTBRANCH is made for each transaction in $head$'s list of successor transactions. Since DEEPESTBRANCH is only invoked on the successors of $head$, and each list in the adjacency list of transactions is finite, it is guaranteed that every invocation of DEEPESTBRANCH will eventually reach a transaction with no successors. Upon reaching a transaction with no successors, DEEPESTBRANCH terminates on line 24 of Algorithm 5. □

Results

This section shows experimental results of tests of the HMS algorithm on a private Ethereum blockchain. The chain used for testing is a fork of an open source multi-peer private network configuration [6]. Experiments were hosted on Ubuntu 16.04 EC2 servers in the AWS cloud. The private network was configured to be a model of the Ethereum mainnet or the Ropsten testnet. Proof of work was used as the consensus mechanism. The block difficulty, transaction fees, processing power of the peers and peering topology were adjusted to produce block size and interval in the range of production Ethereum blockchains.

Interoperability was tested by running experiments with a mix of peers running standard Geth and modified Sereth clients. The first experiments were qualitative to demonstrate practical use of the two innovations of this approach: HMS and RAA. Smart contract functions that created transactions were followed through the process of invocation, interpretation, transactions sent to the TxPool, replay, mining and validation. The Sereth client operated interchangeably with Geth clients on the same network. This is not surprising because Ethereum already supports a variety of clients with subtle differences, all following the same protocol. Deployment of Sereth in the wild would not require a fork or any special permission from the network. The Solidity smart contract equipped with RAA also functioned even when deployed to a Geth client, although of course the substitution of arguments did not take place and they were returned unchanged.

Next we demonstrated that a sequential history was properly handled by sending a series of test transactions from the address of a single peer so that there is only one possible history, where real time order equals nonce order equals block order. As expected, the transaction failure rate was zero and the transaction efficiency η was 1.0.

The quantitative experiments using concurrent peers demonstrated the effectiveness of HMS and

the importance of transaction efficiency. Experiments considered the history of program execution on a single shared variable P where P is an object containing the AMV tuple described in the HMS algorithm. The dynamic pricing exchange from Section 3 (*Challenging Use Cases*) is used to motivate the experiments, with the value of P representing the price. Two transaction types are used in the experiments: buy (buys one item at the current price) and set (changes the price). A ratio of buys (READ-UNCOMMITTED) to sets (WRITES) was used as a non-dimensional parameter that would scale up to larger servers as the absolute number of transactions increased. The number of set transactions was varied from 100 to 5, yielding a buy to set ratio of from 1:1 to 20:1.

Figure 3.2 depicts a plot of state throughput measured at different buy set ratios. Each data point represents the result of 100 buy transactions, so state throughput is equivalent to η expressed as a percentage. Transactions were submitted at an interval of one second, resembling a moderate throughput smart contract use case. This interval was sufficient to demonstrate the problem of stale reads and can easily be reproduced with ordinary servers using the provided source code. The sets are evenly spaced over the processing of the buys. The lines are smoothed averages of the points shown, with the shaded areas representing the 90 percent confidence interval for the lines. Sets are not plotted, as their success is guaranteed.

Standard Geth client

The baseline scenario sends transactions to an unmodified peer running the standard Geth client. The transaction efficiency at different buy to set ratios is labeled as "geth_unmodified" in Figure 3.2. In this scenario, buy transactions that read the price P from block $n - 1$ and are included in block n before the price is modified will be successful, while all other buys will fail. When there are many price sets, as in the experiments with 1:1 and 2:1 ratios, only a few buys are successful. In some runs no buy transactions succeeded at all. The efficiency increases somewhat as the ratio

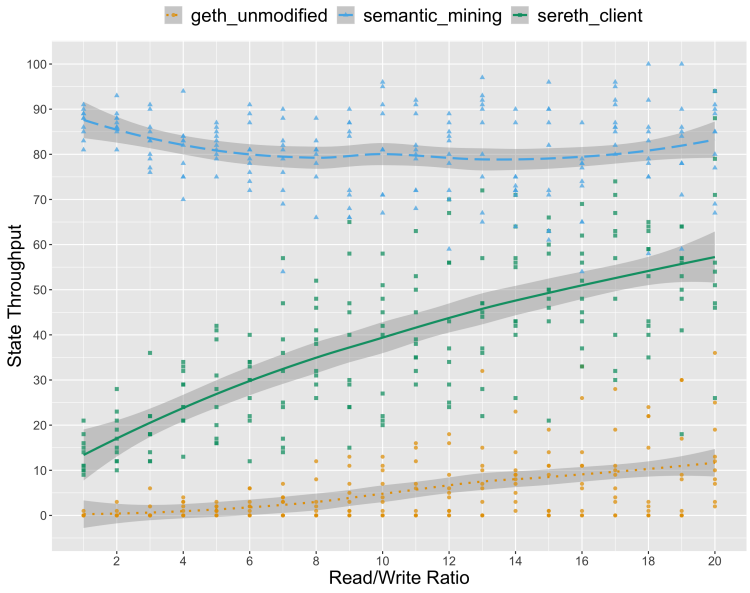

Figure 3.2: Transaction efficiency η vs READ-UNCOMMITTED / WRITE ratio.

of buys to sets goes above 10:1 because there are more buys reading correct values before an intra-block set occurs. However it remains poor for two reasons.

First, with a low ratio of buys it is unlikely for a buy to land in the very beginning of the block before any sets take place. Thus many will fail. Second, even as the ratio increases, because of the large transaction pool there are often no buys going into block $n + 1$ that have a valid view of block n. Instead, block n is assembled from buys that were submitted a few blocks ago, so they may have a view of block $n - 2$, $n - 3$ or older blocks. These buys fail because the blockchain state has passed them by before they were included. This phenomenon is frequently observed in public blockchains [12].

Although not shown in the plot, it was also observed that with few state changes (high ratio of buys) transaction efficiency becomes more sensitive to the transaction interval, as miners may sequence a large number of buys together.

Hash-Mark-Set without miner assistance

The second experimental scenario, labeled as "sereth_client" in Figure 3.2, used the modified Sereth client on the network, implementing the HMS algorithm. The set transactions were ordered with HMS while buy transactions were sent exactly as in the baseline scenario. Interleaved with the sets, any buy at the right mark and price succeeded. The benefit of HMS in this scenario is that the buy transactions have a READ-UNCOMMITTED view of the likely state of the storage variable P when they will be evaluated. This allows many more transactions to succeed. A sequentially consistent ordering of the set operations was established and their dependent buys have a view of the state provided by HMS. Figure 3.2 shows an improvement in throughput by approximately a factor of five over the entire range of read /write ratios. These results were achieved without miner assistance, so they could be accomplished simply by running the modified client on the public Ethereum blockchain, as long as access to the smart contract was via these clients.

This experiment also demonstrates how HMS alleviates the intra-block lost update problem. The FPV arguments in each buy include the previous mark, a hash that relates it to an interval between two sets. If a sequence occurs such as: set(5), buy(5), set(7), set(5), buy(5), a particular buy(5) can prove that it was sent during the first or the second interval the price was set to 5. Linking each buy transactions to a particular set price prevents the frontrunning attack mentioned in Section 3 (*Challenging Use Cases*).

Hash-Mark-Set with semantic mining

In the third experimental scenario, the inputs of the second scenario were repeated with the miner using the HMS algorithm to determine the block order of transactions. In this scenario HMS information about the TxPool is available to both smart contract users and miners. Since the miner

now has awareness of the semantics of the transactions, we call this *semantic mining*. Previously miners would not reorder transactions to increase transaction efficiency, but the semantic miners have this capability. The line labeled "semantic_mining" in Figure 3.2 shows the results. About 80 percent of transactions succeed due to semantic mining providing interleaving in conformance to the READ-UNCOMMITTED view used by the smart contract clients when they sent the transactions. Relative improvement in throughput was greatest with a high frequency of price changes, i.e. where there are 1 or 2 buys per set. At these ratios the advantage of having the miner interleave transactions increases transaction efficiency from a few percent to almost 90 percent, resulting in a factor of six over the unassisted case. Overall, 10-20 percent of transactions were lost due to the fact that the TxPool no longer contains marked transactions immediately after the block is published. Transaction efficiency could approach 100 percent if HMS were extended to include the final values from replaying each block. Other factors that would impact efficiency is if only a fraction of the miners were assisting, or if communication of the TxPool were impeded among the Sereth enabled peers. Performance would be degraded in these cases but there would still be benefits proportional to the participation.

Chapter Summary

State throughput, the throughput of successful transactions, is proposed as the appropriate metric for smart contract performance. An algorithm, Hash-Mark-Set, and a novel architectural technique, Runtime Argument Augmentation, are presented and demonstrated together on the Ethereum blockchain to improve state throughput.

HMS provides smart contracts a READ-UNCOMMITTED view of state. At the same time, HMS provides information about transaction dependencies to the miners so they can adjust the block order, called semantic mining. Miners cooperating with smart contracts using the HMS algorithm

to order dependent transactions were able to create blocks in which most transactions were successful. This is demonstrated to improve transaction efficiency from less than 5 percent to over 80 percent in cases where state changes are frequent, more than an order of magnitude improvement in state throughput. Even without semantic mining, the READ-UNCOMMITTED view is helpful, increasing state throughput by a factor of five across the full range of tested read to write ratios from 1:1 to 20:1. Latency (as a function of correct reads) was also reduced in both scenarios, client modifications only and semantic mining. In addition to the performance gains, HMS solves the blockchain lost update and frontrunning attack problems because transactions using READ-UNCOMMITTED values keep a unique hash validated record of the particular interval during which the value was read.

RAA is presented as a new technique to provide smart contracts rapid communication with external data services. In our experiments smart contracts used RAA to access READ-UNCOMMITTED views of data necessary for transaction success and thus increase transaction efficiency. RAA works at the architectural level of the EVM, using the interpreter to achieve high performance. Peers running the RAA modified client were demonstrated to work interoperably with standard peers.

CHAPTER 4: BLOCKCHAIN SCALABILITY WITH PROOF OF DESCRIPTOR

In this chapter, we present Proof of Descriptor (PoD). We begin by describing the problem of blockchain consensus, and the necessity of a concurrent, scalable solution. We then describe our solution, a consensus mechanism designed using traditional SMP techniques to enable concurrent updates to a shared ledger by multiple network participants. Finally, we provide a performance analysis of our approach using a simulated decentralized network.

Introduction

Several works [25, 49, 21, 9] address the inherent sequential execution of Proof-Of-Stake (PoS) through *sharding*, a technique that partitions the clients into multiple smaller groups of clients called *shards* that operate in parallel on disjoint blocks of transactions and maintain disjoint ledgers. Atomicity and isolation for cross-shard transactions [21, 9] are handled using two-phase locking in conjunction with a *Practical Byzantine Fault Tolerance* (PBFT) consensus [5]. PBFT incorporates three phases: 1) the leader broadcasts the requests, 2) the clients agree on the order of the requests, and 3) the clients commit the requests in the agreed order. In a traditional crash-failure model, a *faulty* client (or thread) is one that stops sending and responding to requests. A Byzantine failure model assumes a more hostile environment where a faulty client may also be a malicious attacker. The challenge with PBFT is that a faulty leader must be replaced with an expensive *view-change* protocol that changes the leader using $O(N^2)$ messages among N clients [9].

We observe that the challenges of providing scalable transaction execution in blockchain is analogous to the same challenges in shared-memory multiprocessor programming, in which threads

synchronize to execute transactions concurrently on a shared data structure. We draw on this extensive body of work to address the shortcomings of PBFT with a cooperative strategy among clients that does not require a designated leader. In multiprocessor programming, a concurrent history of transactions is a sequence of invocation and response events. Two histories are *equivalent* if they contain the same set of transactions, and have equivalent start and end states. A concurrent history is strictly serializable [33] if it is equivalent to some legal sequential history, obeying any real time orderings [19].

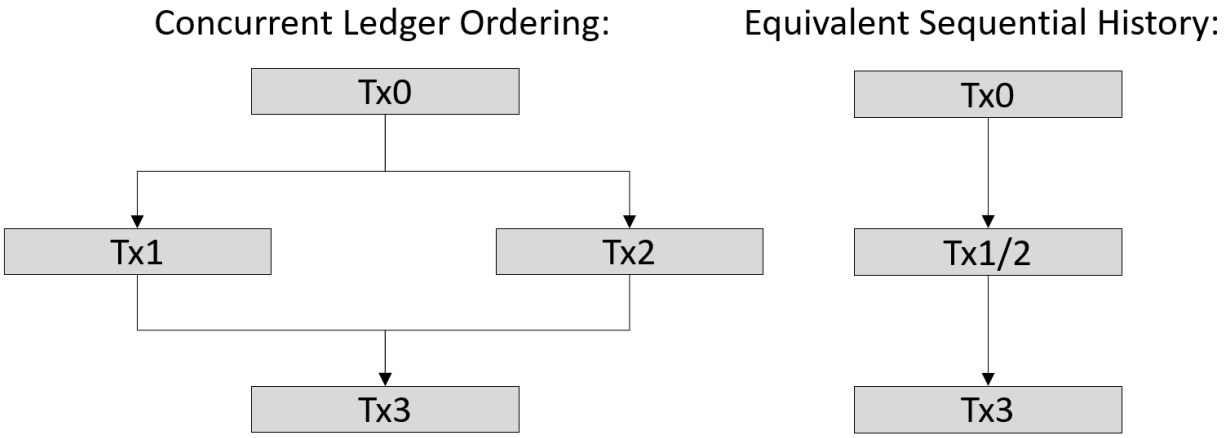

Figure 4.1: Assume Tx1 and Tx2 have no conflicts, and therefore commute. In the equivalent sequential derived from the concurrent ledger order, transactions t1/t2 can be executed in either order to achieve a deterministic final state.

We propose Proof of Descriptor (PoD), a *concurrent* consensus mechanism for decentralized networks. In PoD, clients execute transactions in a strictly serializable manner. The execution order of conflicting transactions is preserved in the form of a Directed-Acyclic-Graph (DAG), which serves as the network's shared ledger. This ledger can be treated as a precedence graph that is conflict-serializable. As such, given a concurrent ledger h produced by PoD, all network participants will observe the same *final* state after executing h from an identical starting state. This solution leverages the fact that two non-conflicting transactions, $t1$, $t2$, can be appended to the

ledger concurrently without a total ordering. If two transactions can be executed in either order without changing the final state of the execution, those transactions are said to commute [19]. This is visualized in figure 4.1.

We design Proof of Descriptor with the goal of enabling arbitrary network clients to achieve consensus on a concurrent ledger, represented by a conflict-serializable precedence graph. To that extent, we utilize a descriptor-based transaction execution algorithm from shared memory, Lock-Free Transactional Transformations (LFTT) [50]. LFTT achieves strict serializability by detecting semantic conflicts between transactions, and enforcing an order on only those transactions which conflict. Executing LFTT, and creating a record of execution order of all conflicting transactions yields a conflict-serializable precedence graph. Any conflict-serializable precedence graph would satisfy the requirements for a distributed ledger, as executing the transactions in any order while respecting their precedence yields a deterministic final state. As such, all network participants will agree on the final state of the ledger despite possible divergence during intermediate steps. There are several challenges for applying an LFTT-style methodology to blockchain consensus. 1) Clients on the network have no access to a source of shared memory, nor access to shared memory atomic primitives such as compare-and-set. Additionally, 2) unlike threads in a multiprocessor, clients on a decentralized network may behave maliciously in order to manipulate the ledger against protocol.

We address the first challenge by designating the role of semantic conflict detection to clients on the network, which we refer to as "Stakers" (Figure 4.2). A stakers role is to provide a sequential ordering for each operation that makes changes to their staked wallet. To guarantee the chosen ordering is sequential, a wallet can only have one staker at any instance of time. The staker has no access to the private key for the wallet it stakes. An owner of a wallet may choose to act as their own wallet's staker, or relegate it to a third party.

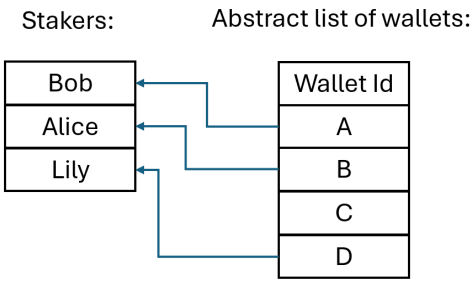

Figure 4.2: Map of stakers to abstract list of wallet addresses.

Given a set of stakers, we can devise a decentralized version of LFTT. When a user submits a transaction to the network, each stakers extracts the list of atomic operations that compose the transaction. A transaction that transfers money from $Alice$ to Bob would be composed of two operations, the first operation removes some quantity from Alice's balance, and the second operation adds that quantity to Bob's balance. For each operation in the transaction, the corresponding staker broadcasts a descriptor object which details that operation's precedence order relative to every other conflicting operation in the ledger. Operations only conflict if they make changes to the same wallet address, thus, any operation that modifies wallet x will eventually be processed by the staker for wallet x. This guarantees that conflicting operations are always handled by the same staker. As such, a staker can guarantee that all operation conflicting about their wallet have a total ordering. Furthermore, a staker is always able to propose an ordering that satisfies the commutativity isolation property of strict serializability. Commutativity isolation states that non-commuting transactions appear to take place sequentially, without interleaving. If a staker broadcasts a descriptor $n1$ as part of a transaction $t1$, then they will order the subsequent descriptor $n2$ (part of $t2$) *after* the completion of $t1$. This results in a history where $t1$ *appears* to execute completely prior to the start of $t2$. By broadcasting a descriptor, a staker is making a *logical* update to the shared database, as the descriptor contains all information about how to apply its underlying operation. This ordering between descriptors is communicated through the use of cryptographic hashes, simi-

lar to the cryptographic linking of blocks in block-based ledgers. If a staker broadcasts a descriptor $d2$ after a prior *conflicting* descriptor $d1$, it will include that hash of $d1$ in $d2$. The state of the ledger can be computed by treating the graph of descriptor objects as a precedence graph, executing each operation for each transaction in the order given by the descriptors. This precedence graph is conflict-serializable, as LFTT's semantic conflict detection ensures that all conflicting transactions have a total order. This ensures that all participants will compute the same final state.

To address the second challenge, we introduce a proof of stake scheme to discourage misbehavior by stakers. It is possible for a staker to lie about the ordering of their descriptors in an attempt to retroactively modify the ledger. However, this behavior is impossible to hide, so long as a record of all descriptors broadcast by that staker are available in the ledger. If a staker broadcasts a descriptor that does not have a total ordering with every other descriptor they have broadcast, the staker is slashed, and their stake is lost. A staker cannot feign ignorance in this situation, as the staker is guaranteed to be aware of any descriptor they themselves created. The only exception for this case is if a staker is making a correction to their own history as a result of faulty descriptors created by a *different* staker on the network.

Similar to block based approaches, each descriptor includes the hash of any prior descriptor objects that it references. As the ledger grows, older descriptors gains many descendants. Changing the contents or ordering of a descriptor would change the hash of all descendants of that descriptor, many of which will belong to other stakers on the network. As such, a staker can only succeed if the staker of every descendant descriptor joins the attack.

Figure 4.3 gives an overview of the concurrent ledger. In this figure, an arrow from descriptor n to descriptor m indicated that n was created before descriptor m. Consequently, n should be executed before m when computing the state. The state of the ledger can be computed by executing each transaction in the order given by a reverse topological sort of the graph. This topological sort is

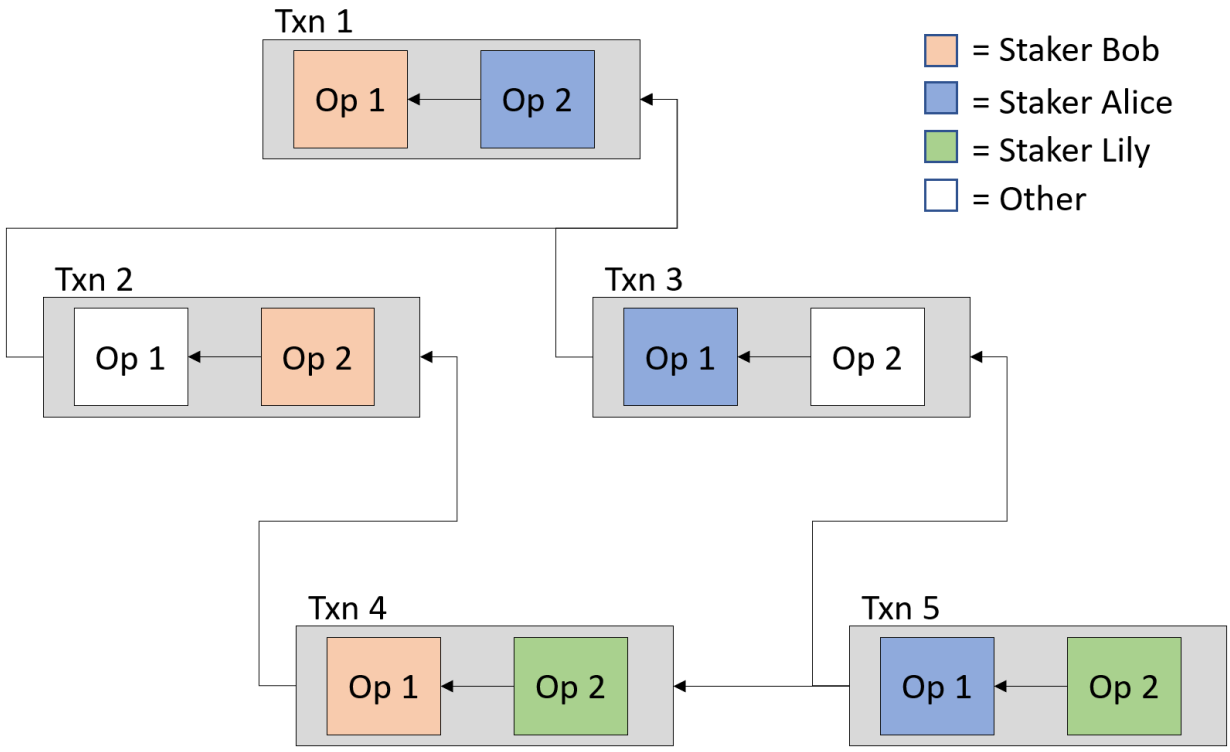

Figure 4.3: An overview of the ledger. Descriptor objects correspond to transaction operations, which make an atomic change to some wallet address. Descriptors produced in sequence by the same staker have a total ordering, because they directly conflict on a wallet address. The state of the ledger can be computed by executing transaction in the order given by a reverse topological sort of the graph.

guaranteed to produce an order that is equivalent to some sequential ordering because the ledger is conflict-serializable. Furthermore, the final computed state is deterministic, as any two transactions without a total order (such as transactions 2 and 3) are guaranteed to commute, therefore, their relative order does not affect the final state. If two transactions do not commute, it means they *must* share at least one staker, in which case that staker will determine the relative order of the non-commuting transactions.

Motivation

Blockchain technology is only a decade old, but is already bruited to replace traditional financial systems [40]. Public blockchain implementations provide the advantages of trust [10], resilience [42], financial inclusion [23] and decentralization [45]. Scalability is a major hurdle to adoption. Decentralized public blockchains in wide use (Bitcoin, Ethereum) have a throughput of $O(10^1)$ transactions per second (tps) [2]. This is not adequate for the use cases at the core of the modern financial system such as large E-commerce websites $O(10^3)$ tps [35], card payments at peak times $O(10^4)$ tps [43] and major stock, option and FX exchanges $O(10^5)$ tps [7].

Methodology

In this section, we detail the methods used by stakers to process transactions and publish them in the ledger.

Algorithm 6 Definitions

```
 1: struct Staker
 2:     int stakedWallets[]
 3:     Map<int, Descriptor *>descriptors
 4: struct Operation
 5:     int type
 6:     int value
 7:     int address
 8:
 9: struct Transaction
10:     int size
11:     Operation ops[]
12: struct Descriptor
13:     int opNumber
14:     Hash transaction
15:     Hash prevOpDesc
16:     Hash prevTransactionDesc
17: struct Graph
18:     Descriptor *root
19:     Map<int, Descriptor *>walletMap
```

Algorithm 6 details the main structures for our algorithm. Here, we list only the objects pertaining to the core functionality of PoD, as implementation details may vary.

The STAKER object contains fields used by a staker to track the wallets they are currently staking,

as well as the descriptors they have created for those wallets. The stakedWallets field is an array of wallet addresses that a staker is currently staking. The staker will be able to broadcast descriptors for operations that affect any wallet within the stakedWallets array. This does not give the staker the ability to create new transactions that affect these wallets unless they are the owner of the wallet. A wallet can only be staked by a single staker on the network at a time. In the case that the owner of a wallet wants to designate a new staker for their wallet, the wallet must be transferred between stakers as part of a transaction, as this ensures the transfer occurs in a single atomic step with a verifiable ordering relative to other transactions. The descriptors field maps wallet addresses that are currently being staked to the most recent descriptor created for that wallet.

The OPERATION object stores information about the individual operations that make up a transaction. The type and value field define the behavior of the operation, while the address field identifies the address of the client whose wallet will be affected by the operation. These fields can vary depending on the type of operation being performed.

The TRANSACTION object contains a sequence of operations which make some change to the state of the ledger. Similar to traditional blockchain designs, a transaction is created when a user wants to transfer some value between wallets. A transaction also contains the transaction signatures necessary to verify the identity of its participants. Transactions acts as the blueprint for stakers to create descriptors. The transaction object is passed alongside any descriptor objects to stakers so they can view the operation affecting their staked wallet.

The DESCRIPTOR object is used to resolve conflicts between transactions that affect the same wallet. A descriptor contains several hashes corresponding to the transaction, and operation that it represents, as well as the preceding, conflicting descriptor *prevTransactionDesc*, and the previous operation belonging to the same transaction *prevOpDesc*. The job of creating a descriptor, and therefore ordering a transaction relative to another, conflicting one, is performed by the cur-

rent staker for the affected wallet. This means that two conflicting operations will necessarily be processed by the same staker, eventually. In a typical multicore algorithm, a descriptor would generally be stored as a reference within the object being modified (such as a node). In our algorithm, descriptors are stored in the ledger, which is replicated across all clients.

The GRAPH object represents a client's local copy of the distributed ledger. The relationship between descriptors by their *prevOpDesc* and *prevTransactionDesc* hashes naturally form a DAG. The root node of our graph is analogous to the genesis block in PoW or PoS. It represents the initial state of the ledger, in which no wallets have been created, and no state changes have occurred. All descriptors will descend from the root node. The walletMap maps a wallet address x to the last descriptor of the most recent transaction that affected wallet x. Throughout our pseudocode, we assume that stakers update their graph in the background, in response to descriptors received from other nodes on the network.

Algorithm 7 Staker Descriptor Processing

```
1:  function PROCESSDESCRIPTOR(Transaction *t, int opNum,
        Descriptor * descriptors)
2:      walletAddress ← t.ops[opNum].address
3:
4:      Desc *d ← Create descriptor from t
5:      d.prevOpDesc ← descriptors[opNum-1]
6:      d.prevTransactionDesc ← Graph.walletMap[walletAddress]
7:
8:      if prevTransactionDesc.t.status != ACTIVE then
9:          d.hash ← sha256(d, prevOpDesc, prevTransactionDesc)
10:         this.descriptors[walletAddress] = d
11:         signAndEmit(d)
```

Algorithm 7 details the pseudocode for broadcasting descriptors. The goal of all stakers on the network is to broadcast descriptor objects in an order that is strictly serializable, and to capture this ordering to be preserved in the shared ledger as a precedence graph.

Upon receiving a transaction t from the mempool, a staker creates a new descriptor d. The staker

is only concerned with the operation from *t* that modifies their staked wallet. The staker searches for two distinct descriptors, which must exist before descriptor *d* can be published in order to guarantee strict serializability.

- The "previous descriptor" *prevOpDesc* is the descriptor for the previous operation in transaction *t*. If the current operation is the first operation in the transaction, then *prevOpDesc* is left set as $null$, since there is no previous operation to reference. By waiting for this descriptor, the staker ensures the transactions operations are applied sequentially.
- The "previous transaction" *prevTransactionDesc* is the final descriptor of the previous transaction that modified this staker's staked wallet. By waiting for this descriptor, the staker satisfies the commutativity isolation property of strict serializability. Stakers will always know which transaction this is, as a prior descriptor they created will be a part of it.

Once the staker has collected the necessary descriptors, it computes the hash for *d* on line 9. This hash is used to prove to any arbitrary client that both *prevTransactionDesc* and *prevOpDesc* existed when descriptor *d* was created. This hash serves to strengthen the immutability of the ledger, as well as to indicate the precedence order of a descriptor within the ledger. On line 11, the descriptor is signed by the staker. Once the descriptor is fully formed and signed, it can be emitted to the network. Doing so prompts the staker for the next operation in the transaction to execute *processDescriptor*. If all stakers have executed *processDescriptor*, the transaction is complete. For the purpose of correctness, a transaction's linearization point occurs when the final staker executes line 11, broadcasting their descriptor.

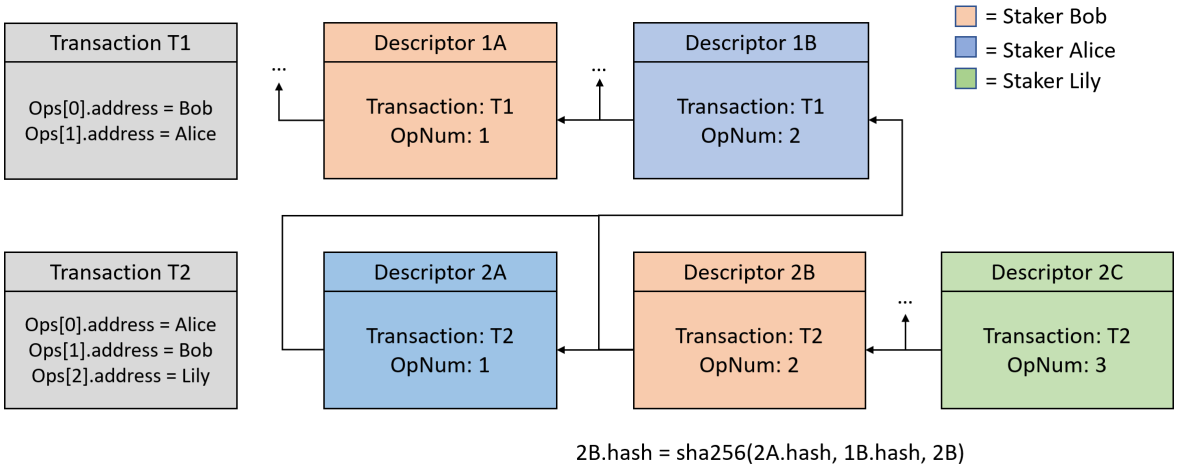

Figure 4.4: Hash calculation for descriptor objects.

Directed Acyclic Graph

The hash of each descriptor object is generated based on a PoH scheme to prove its order relative to other key descriptors. PoH leverages the observation that by calculating a hash *H1* with a secure and collision-resistant hashing algorithm to produce a hash *H2*, it can easily be proven that hash *H2* was computed after *H1* since *H1* must exist to compute *H2*. Furthermore, an additional hash *H1a* could be appended to *H1* before it is hashed into *H2* to prove that both hashes *H1* and *H1a* were computed before *H2*. We use this to commit a precedence ordering between descriptors to the ledger.

The hash of each descriptor object *m* is calculated from three components. The first component is the hash of the descriptor object that comes before *m* within *m's* transaction. The second component is the hash of the final descriptor object for the transaction that previously accessed the same wallet as *m*. The final component is the contents of the descriptor *m* itself. We refer to any descriptor *d* whose hash is used to calculate *m* as a parent on *m*. We choose the parents of *m* such that their

ordering satisfies strict serializability. Specifically, they prove that a transaction's operation took effect in sequential order, and that any prior conflicting transaction appeared to execute completely before the next one began.

Figure 4.4 is a case with three stakers, Bob, Alice, and Lily. Each staker has a single unique staked wallet. In this example, there are two transaction composed of two and three operations respectively. Arrows point from child descriptors to their parents.

We can reason about the ordering of these transactions given just their parent-child relationships. In this case, we must conclude that transaction 1 was committed before descriptors *2A* or *2B* were created, since their hashes are computed from the descriptors of transaction 1. If any staker in this system had attempted to violate transaction isolation by creating descriptors for multiple active transactions, those inconsistencies would be visible in the resulting graph in the form of a cycle, or an invalid parent-child relation. The ordering formed between these transactions is difficult to modify, as in order to change a descriptor, one must recompute all descendants of that descriptor. For this reason, a descriptor acts as a "confirmation" of all parents and grandparents. In order for *Bob* to modify descriptor *1A*, *Bob* would have to recompute the hashes for descriptors *1B, 2A, 2B* and *2C*. So long as *Alice* and *Lily* are behaving honestly, *Bob* is unable to modify descriptors *1B* and *2C* without the private keys for *Alice* and *Lily*.

Thus, descriptors created by PoD form a Directed Acyclic Graph (DAG) structure via their hashes. In this graph, each vertex is a descriptor object, and each edge represents a parent-child relationship, in which the hash of the parent node is used as input for the hash of the child. This graph constitutes the shared ledger, as the combined descriptor objects contain enough information to reason about the order of any transaction with respect to any other transaction non-commuting transaction. Transactions are executed from parent to child, starting from the root node. A property of our approach is that whenever a descriptor from wallet x becomes a descendant of a descriptor y, all

subsequent descriptors from any transaction that affects wallet *x* will also be descendants of *y*. This property enables descriptors to eventually be confirmed by the majority of network participants.

Staker Transferal

A wallet owner may sometimes want to change the staker of their wallet. To do so, a "transfer" transaction must be issued containing the identity of the new staker. This transaction is processed like all others, and must have a valid *prevTransactionDesc* field to establish the order in which the transfer took place relative to other transactions.

Security

In this section, we provide a proof that a ledger generated by our approach satisfies strict serializability so long as clients are operating in accordance with the consensus protocol. We proceed to analyze possible attacks on the network, and provide remedies in line with existing protocols like Proof-of-Stake.

Safety

The presented formal proof of safety leverages the work of Herlihy et al. [17]. A *history* is a sequence of invocation and response events. A *sequential history* is a history in which each event has an instantaneous response. Two operations *commute* if they access different wallet addresses.

Definition 1. A history h is *strictly serializable* if the subsequence of h consisting of all events of committed transactions is equivalent to a legal history in which these transactions execute sequentially in the order they commit [33].

Definition 2. *Commutativity Isolation. For any non-commutative operations, $op_1 \in T_1$ and $op_2 \in T_2$, either T_1 commits or aborts before any additional operations in T_2 are invoked, or T_2 commits or aborts before any additional operations in T_1 are invoked.*

Lemma 3. *Proof of Descriptor satisfies commutativity isolation.*

Proof. Let T_1 denote an active transaction that modifies wallet w_1, i.e., $op_1 \in T_1$ and $op_1.address = w_1$. Assume W_1 is currently staked by staker s. If another transaction T_2 were to attempt to modify w_1, it will eventually arrive in the PROCESSDESCRIPTOR(T_2, n, d) method of staker s. The staker will discover that the $s.descriptors[w_1]$ field contains an already active descriptor for wallet w_1, on line 8. Since the if-statement resolves to $false$, a descriptor for T_2 will not be created, and T_2 will need to wait until T_1 is complete prior to proceeding with the operation on w_1. When T_2 retries an access to w_1, a descriptor will be created if T_1 has been completed and no other transactions have been accepted since then. It is thus ensured that the descriptor stored at $s.descriptors[w_1]$ is committed before T_2 proceeds. □

Definition 3. *A history h is linearizable if the method calls take effect in real time order, and given a method call with invocation I and response R and another method call with invocation I' and response R', two concurrent invocations I and I' must be equivalent to either the history $h \cdot I \cdot R \cdot I' \cdot R'$ or the history $h \cdot I' \cdot R' \cdot I \cdot R$.*

Lemma 4. *The* PROCESSDESCRIPTOR *operation executed using the Proof of Descriptor protocol is linearizable.*

Proof. The PROCESSDESCRIPTOR operation modifies the state from the wallet's current value. Recall that stakers assume the role of Compare-And-Swap. A descriptor object contains all information necessary to fully describe an operation that makes some state change to the network. Thus,

this update is performed logically as soon as a descriptor is broadcast by a staker. The linearization point, i.e. the instant in which the PROCESSDESCRIPTOR operation takes effect, occurs when the staker locally updates their descriptor pointer on line 10. Since the wallet's value is updated in one indivisible step at some instant between the PROCESSDESCRIPTOR operation's invocation and response, PROCESSDESCRIPTOR is linearizable. □

Theorem 1. *For a block of transactions that is executed using the Proof of Descriptor consensus protocol, the history of committed transactions is strictly serializable.*

Proof. Follow lemmas 3, 4, and the main theorem of Herlihy et al.'s work [17], the theorem holds. □

Threat Model

The model comprises a peer to peer message passing communication between clients. The communication is asynchronous, which implies that descriptor delivery time can be infinitely delayed. All descriptors are required to be 1) cryptographically signed by the originating staker, and 2) made publicly visible to all clients. A staker is a client that owns some minimum threshold of monetary stake on the network, and has 0 or more staked wallets. Wallets cannot be modified without an active staker. The model assumes an honest majority of stakers, meaning $> 50\%$ of stakers act in accordance with the agreed upon protocol. Transactions include a list of all wallets that will be modified during execution. Stakers create descriptors for the purpose of ordering conflicting transactions. Stakers automatically reject transactions that are not signed by all involved wallet owner, or are malformed.

Transaction Immutability And Finality

Transaction's "lazily" gain confirmations over time based on their semantic conflicts. When a staker submits a descriptor d, that staker implicitly commits to a view of the ledger that includes all parent descriptors of d in the precise order given by the hash of d. This is demonstrated in figure 4.5.

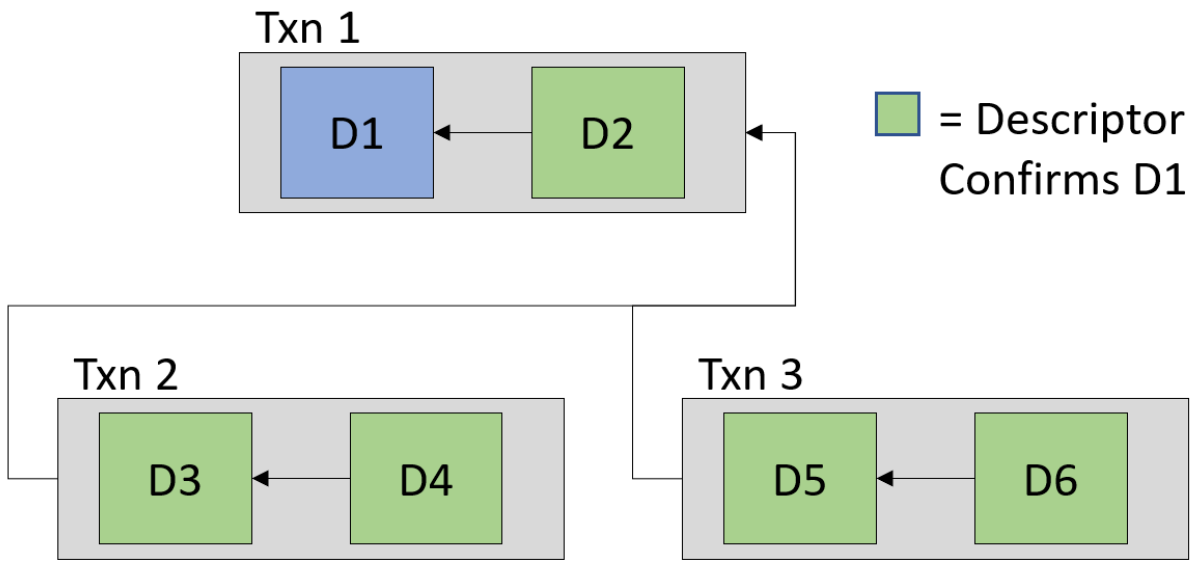

Figure 4.5: Descriptors contain the hashes of their parents. These descriptors are said to "confirm" their parent. Over time, a descriptor (D1) gains many confirmations, as its number of descendants grow.

If an attacker were to attempt a long range attack by modifying a descriptor d, the attacker would also need to recompute the hash of all descendants of d. Although this does not require a large expenditure of computation, like that of PoW, it is still infeasible if the modified descriptor has a large body of confirmations. This is because a descriptor is invalid if it is not broadcast by the unique staker for the wallet address modified by d at $d's$ location in the graph. As such, if d has any descendants that were produced by a staker other than the attacker, the attacker would need to

either forge the signature of that descriptor, or convince the staker to join the attack. If the attacker cannot do this for *every* descendant of d, the attacker's corrupt ledger would contain at least one pair of descriptors with mismatching hash sequences. PoD is similar to related approaches in that the chance of a successful long range attack on a transaction decreases for transactions with more confirmations. Specifically, a transaction that has received confirmations from $x\%$ of stakers on the network would require a coordinated attack from all $x\%$ of confirming stakers to modify. As such, it is possible for a transaction to achieve 100% immutability if it has received confirmation from 100% of the staker on the network.

Since relations are only formed between transactions that have a semantic conflict, a particular descriptor could remain in an isolated section of the PoD graph if it is part of a wallet with a low volume of transactions. We address this problem by optionally including some number of "auxiliary parents" with descriptors. These descriptors have no semantic conflict with their auxiliary parents, but still provide a confirmation to the auxiliary parent by including its hash and acknowledging its correctness.

Illegal Fork

The *prevTransactionDesc* field of every descriptor created by a staker *s* form a sequential chain, showing the relative ordering of every transaction involving the wallet staked by *s*. This sequence allows the state of any wallet in the network to be computed by applying transactions in the order formed by their descriptors. However, a malicious staker could select a *prevTransactionDesc* field that does not truly reference the most recent transaction involving *s*, instead referencing some older transaction. Figure 4.6 demonstrates this case. Here we see that a fork is created after Descriptor 1b by Bob. Since there is no total ordering between descriptors 2a and 3b, it is unclear in which order they should take effect. The separate branches can never be merged, as the resulting history

would not be serializable. Thus, one branch must be ignored. A malicious staker may attempt this in an effort to "undo" a transaction, by convincing the network to commit to a new branch that doesn't contain the transactions they want undone.

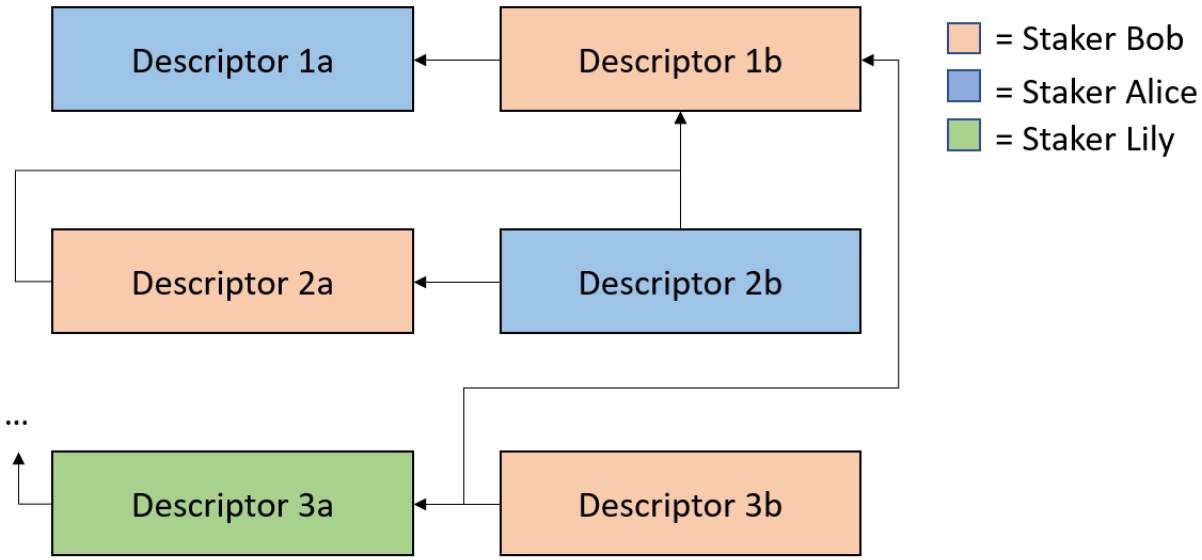

Figure 4.6: Descriptors 2a and 3b have no total ordering, despite conflicting about Bob's staked wallet. Since all conflicting descriptors in the ledger must have a total ordering, one branch must be ignored. This is an attack by Bob in an attempt to undo either Descriptor 2a, or 3b, and is analogous to a double-spend.

In PoD, we mitigate this attack by having each staker append their staked currency to the ledger as a transaction. These "stake" transactions must be appended every *n-th* descriptor emitted by the staker. This transaction is included in the descriptor sequence. In the case that an attacker creates a fork, it can be ambiguous which branch is the original, and which one is new. However, since stakers include their stake in the descriptor sequence of their descriptors, and since each descriptor confirms its parents, we can assess the stability of a descriptor by the sum of staked currency that confirms it. Consequently, in the case that an illegal fork appears in the ledger, we choose to ignore the branch with less staked currency confirming it. When a transaction is newly added to the

ledger, it can quickly start receiving confirmations from subsequent transactions, some of which will include some amount of stake. The more stake that confirms a descriptor, the more difficult it becomes to undo the descriptor by creating a competing a branch. In order to do so, the attacker must control more stake than the growing number of stakers on the original branch. This is similar to PoW or PoS, in which users often wait for a certain number of block confirmations before they consider their transaction final.

Once the fork has been resolved, the network must come to agreement on the most recent descriptor from the offending staker in the chosen branch. This can be handled similar to the approach described in Ethereum EIP2982 [13], in which a 2/3 super majority vote to slash the offending staker at a point in time corresponding to descriptor d means the network will ignore any descriptors produced by the offending staker that are descendants of d. In the example given by figure 4.7, honest stakers vote to slash the offending staker at $Descriptor3$, as it is the latest descriptor in the branch with the most staked tokens at the time of the vote.

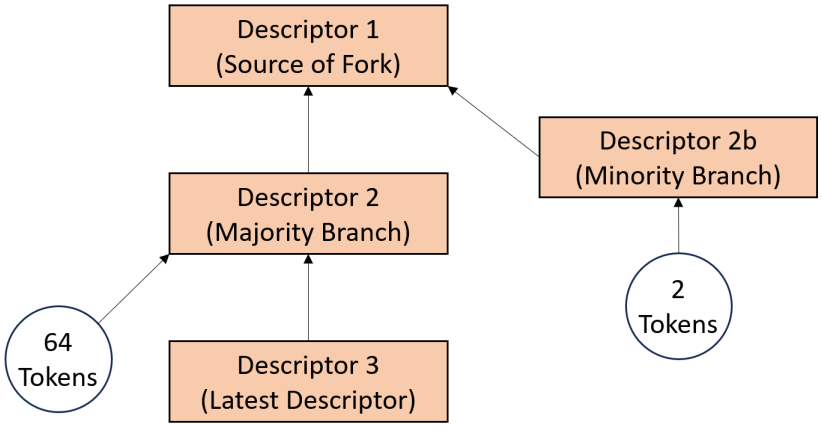

Figure 4.7: A staker produces a fork at descriptor 1. The branch corresponding to descriptor 2 has a larger quantity of network staked tokens confirming it, and is considered the majority branch. Descriptor 2b has fewer tokens confirming it, and is the minority branch. Descriptor 3 is the latest descriptor from the offending staker at the time of the fork.

In the case that a staker is lured into creating descriptors on an attacker's branch, that staker will need to switch over to the branch that has more stake confirming it. This would create a fork in their own history, but should not be considered malicious since it cannot be proven that they were involved in the original, illegal fork. Assume in figure 4.6 that the branch starting with descriptor 3b has more stake confirming it. In this case, Alice cannot continue her descriptor sequence after descriptor 2b, as it is on a branch that will be ignored by the network. In this case, she can ignore descriptor 2b and proceed from descriptor 1a, but ensure that her next descriptor confirms descriptor 3b via auxillary parent. This proves that the fork she creates at descriptor 1a came as a direct result of the fork created by Bob, and that she switched *after* Bob issued descriptor 3b.

In this approach, stake serves two purposes. Firstly, stake acts as a deterrent for malicious behavior, as it is forfeited if a staker creates an illegal fork in the ledger. Secondly, it serves a "vote" on the state of the ledger. By placing stake into the ledger following a particular descriptor, a staker is expressing an intent to acknowledge the full history of transactions that leads up to that point, as well as an intent to continue that history by appending more transactions. The staker is discouraged from switching to a conflicting branch, as doing so would undo any work they did in the original branch.

Transaction Denial Attack

A transaction denial attack occurs when an attacker tries to prevent a certain transaction from being confirmed. This attack can occur if the staker of a wallet ignores a valid transaction, refusing to produce a descriptor and stalling the transaction's progress indefinitely. This situation can also occur if a staker were to go offline without first transferring or relinquishing their staked wallet.

Our algorithm mitigates this attack because descriptors must be created in the same order as the operations for the transaction. Therefore, the next staker for an active transaction is always known.

Stakers on the network wait until they have personally observed inactivity from a staker pertaining to an active transaction for a minimum threshold of time, after which they will initiate a vote to slash the staker using the same approach as the previous section.

Centrality

The scaling problem has been stated as a trilemma: only two of three desired properties (security, decentralization and scalability) can be achieved in a blockchain architecture [46]. Although disputed in theory [26], with some blockchains claiming solutions for specific applications such as bank to bank transfers and centralized program executions[2], the trilemma remains a practical challenge.

In this paper, we proposed a blockchain consensus solution with emphasis on scalability, and security. Our approach does not claim to solve the blockchain trilemma, as there are several factors in PoD that could prevent clients with few resources from competing with clients with many resources. Firstly, PoD stakers must remain constantly active. stakers who cannot keep up with the network will be unable to work, and get slashed. Secondly, in order to minimize latency, stakers with a large quantity of stake may be incentivized to stake many wallets, allowing them to avoid the network latency cost of waiting for other stakers to complete a transaction. Additionally, the increased number of fees collected for processing transactions on a larger number of wallets would incentivize stakers to grow as large as possible. Though we believe these incentives could be mitigated, solutions for doing so are beyond the scope of this work.

Experimental Results

In this section, we analyze the factors which affect the scalability of PoD in order to compute maximum throughput in realistic scenarios. Additionally, we perform an analysis on confirmation growth of transactions for varying numbers of auxiliary parents.

Effect of Latency

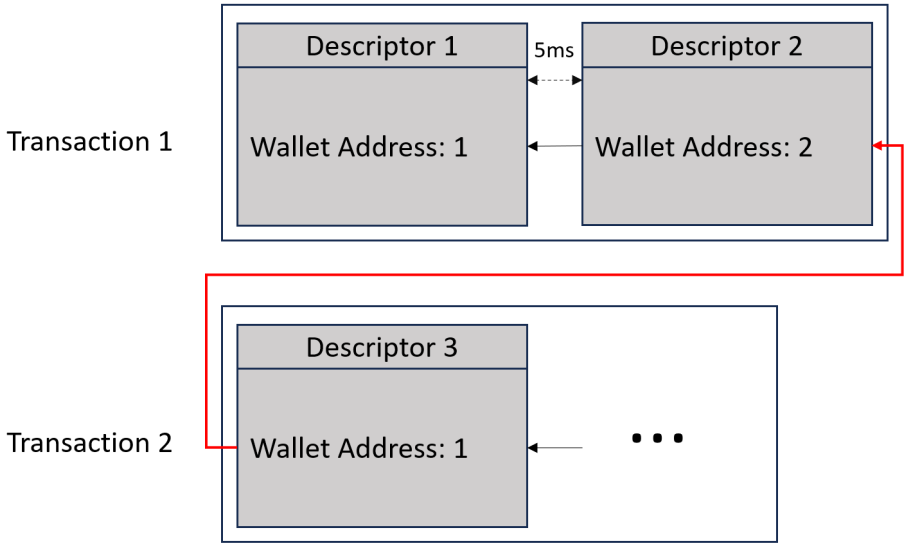

Figure 4.8: Effect of latency on a PoD transaction.

In PoD, stakers work with each other to efficiently build the shared ledger. This can result in situations where a staker must wait on the result of a network communication with another staker. Figure 4.8 demonstrates an example where we are trying to publish two transactions that each begin by modifying Wallet 1. In this case, the staker for Wallet 1 would be responsible for creating Descriptors 1 and 3. In order to ensure an unambiguous transaction ordering, and to preserve commutability isolation, Descriptor 3 cannot be created while Transaction 1 is active. Therefore,

Descriptor 3 must be created after Descriptor 2. If it takes 5ms after the creation of Descriptor 1 to alert the next staker via network communication, then Transaction 2 must wait at least 5ms before it can begin. The wait time associated with P2P latency is typically much larger than the CPU time needed to process a transaction and create a descriptor. This means that the throughput of consecutive non-commuting transactions is limited by the average latency of network communications. Commuting transactions are unaffected by this, as they can execute in parallel. Thus, assuming all wallets are equally active, the throughput in terms of transactions per second for a PoD network scales in two ways:

1. As the average latency of the network decreases.

2. As more wallets are added to the network, increasing the percentage of transactions that commute.

We design our throughput experiments to evaluate our algorithm with respect to these metrics.

Transactions Per Second

We benchmark our algorithm by designing a test to simulate stakers processing transactions on a distributed network to measure transactions per second (tps). In our design, each staker stakes a single wallet, and attempts to execute as many transactions as possible within a timeframe. We restrict each staker to a single wallet to ensure that each transaction requires at least one network communication in order to thoroughly examine the effects of latency.

Stakers execute on a single process and communicate through shared memory with a tune-able communication delay. We compare against the Proof of Work and Proof of History (PoH) [48] consensus mechanisms, where PoH is the consensus mechanism used by Solana [47].

In our benchmark, we measure the time it takes for 1,000,000 transactions to be committed. We select this quantity so that the average experiment executes for about 5 second. Each transaction represent a common blockchain use case, the sending of currency from one wallet to another. Thus, each transaction composed of two operations, one for the sending wallet, and one for the receiving.

Each staker receives a constant influx of transactions initialized by executing their PROCESSDESCRIPTOR method, allowing them to create a descriptor and pass it on to the next staker. This represents a case where each wallet is maximally active, sending as many transactions per second as the network can handle. Each staker tracks their execution time, included time spent waiting for an active transaction to complete. The maximum of these times is used to compute the tps of the network in terms of successful transactions.

Communication Latency

Real blockchains are not fully connected, nor is communication perfectly reliable. Clients self-organize with a limited number of peer connections (Bitcoin uses 8) and do not enforce a network topology. Communication is broadcast to all peers, so the shortest path is found by brute force. We calculate the average minimum hops (shortest path) using the random network model of Erdös-Renyi [15].

$$l_{ER} = \frac{ln(N) - \gamma}{ln(k)} + \frac{1}{2} \tag{4.1}$$

Eulers constant $\gamma = 0.5772$. Average path is proportional to the log of the number of clients over the log of the number of peer connections. In our experiments, we multiply the latency of a single hop by the average number of hops given by equation 4.1 to compute the time penalty of sending a descriptor object to a staker on the network. This means that as the number of stakers

on our network grows, so does the average minimum hops. Our experiments demonstrate that this increase in communication time does not outpace the network's ability to scale with added wallets.

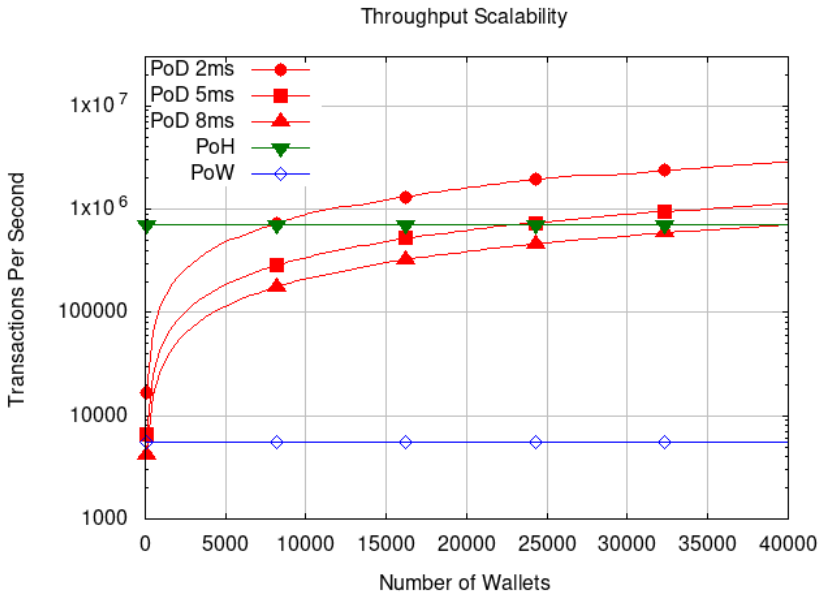

Figure 4.9: Throughput scaling of PoD, compared against PoH and PoW.

Figure 4.9 gives the results of our scalability experiment, plotted using a log scale on the y-axis. For comparison, we plot our results against PoW, maintaining a throughput of 5,500 tps, in line with the peak tps measures in BSV. We plot PoH at 710,000 tps in accordance with their maximum theoretical throughput[47].

We plot 3 experiments for PoD, representing different levels of simulated latency for a single hop on the network. We note that as the number of wallets on the network grows, the average number of hops increases. "PoD 8ms" represents the case where each hop takes 8ms, and therefore the cost of sending a descriptor to the next staker takes as much as 53ms when the number of wallets (and therefore the size of the network) is 40,000. Although the overall latency rises with the size of the network, the scalability generated by adding wallets to the network steadily outpaces the added latency, achieving as much as 2,800,000 tps in the 2ms case with only 40,000 wallets

on the network, and 700,000 tps in the 8ms case. These findings agree with our performance evaluation given by figure 4.8. Firstly, we see that when we reduce the latency of each hop, our throughput increases at all wallet counts. This is due to the fact that stakers are able to more quickly pass their descriptors forward to the next staker, completing each transaction in a faster time. Similarly, increasing the number of wallets on the network increased the percentage of transactions that commute in the case that all wallets are operating at maximum workload. Since commuting transactions are able to execute concurrently in PoD, this increases throughput as wallets are added.

Our approach outperforms Solana's theoretical maximum by as much as 4x in the 2ms case, and in the 8ms case, our approach matches Solana's performance at 40,000 wallets. Solana's performance is unaffected by the number of wallets on the network, as Solana does not allow transactions to be mined concurrently, thus there is no effect of contention. Our results demonstrate that our approach is scalable, surpassing Solana's theoretical throughput in under 40,000 wallets across a range of network latencies.

Transaction Confirmations

In our second experiment, we evaluate the growth of transaction confirmations as transactions are added to the ledger. To do so, we compute the average number of descendants per descriptor in the ledger starting from initialization. Unlike PoW, there is not a large inherent mathematical cost associated with computing the hash of a descriptor. Instead, the hash of a descriptor is composed of the descriptors that lead up to it. For this reason, descriptors are "confirmed" by their descendants. The more descendants a descriptor acquires, the more difficult it is to modify or revert that descriptor.

We prepare our experiment by filling our ledger with transactions using the same distribution as the previous experiment. We begin with the ledger being empty, representing a state where no

descriptors have any descendants, and observe the average number of descendants per descriptor as the size of the ledger grows. We repeat this experiment, varying the number of auxiliary parents per descriptor from 0 to 4 in order to demonstrate the effectiveness of auxiliary parents at increasing the connectivity of the overall graph. The auxiliary parents for each descriptor are chosen randomly from the pool of all descriptors already in the graph. This represents a worst-case scenario where coordination is minimal and clients cannot be guaranteed to choose up-to-date, effective auxiliary parents.

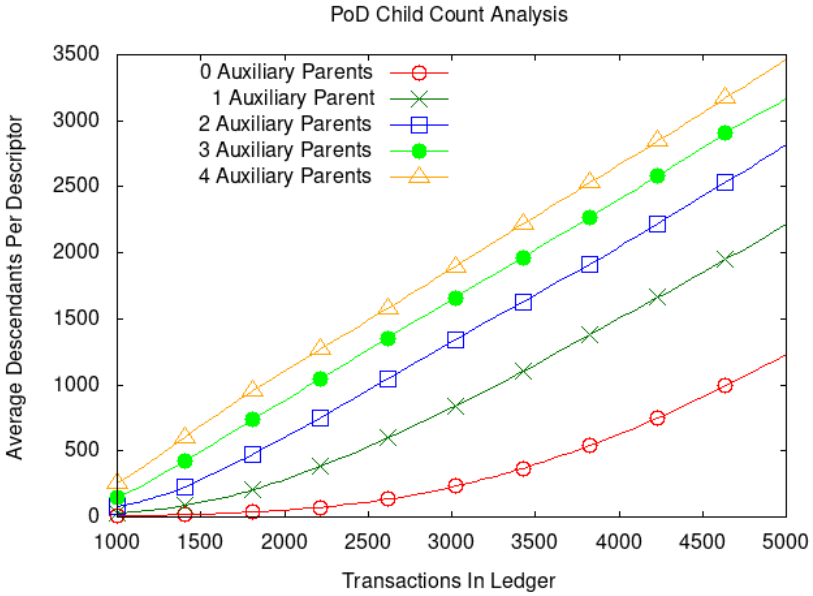

Figure 4.10: Analysis of confirmation growth.

Figure 4.10 displays the results of our security analysis. We observe that as the number of transaction in the ledger increases, the average number of descendants per descriptor increases as well. This growth occurs faster in ledgers where more auxiliary parents are used. In the 4 auxiliary parents case, the average descriptor achieves confirmations from 3,500 descriptors. Since each transaction contains two operations, this represents 35% of the descriptors in the ledger. This rapid growth in confirmation count across all descriptors in the network ensures that any arbitrary transaction can reach finality.

Chapter Summary

In this chapter we described Proof of Descriptor, a descriptor based methodology for solving the consensus problem on a decentralized network. Proof of Descriptor manages shared resources to enable commutative transactions to be appended to a shared ledger concurrently. The use of descriptor objects allows parallel cooperation among stakers, who work to find valid orderings between non-commuting transactions. Written to the ledger, descriptor objects persist to prove not only that a transaction took place, but how and where. Unlike traditional sequential consensus mechanisms, our proposed Proof of Descriptor scales with the size of the network, and inversely with communication latency.

CHAPTER 5: LOCK-FREE CONCURRENT SMART CONTRACTS

In this chapter, we present an approach for lock-free execution smart contracts. A concurrent consensus mechanism like the one discussed in chapter 4 would be costly to implement in existing blockchains. In this section, we describe an approach for existing networks to benefit from concurrency without changing their consensus mechanism. Additionally, we provide an experimental evaluation of our approach in comparison to related works.

Motivation

There are several steps involved in the creation and continued execution of a smart contract. Firstly, a smart contract's code is compiled and submitted in the form of a transaction to the mempool. Miners select transactions from the mempool to include in a block, which they then try to append to the ledger using a consensus mechanism. Once a smart contract has been deployed, its method calls are also submitted as transactions, and included in subsequent blocks. In order to compute the changes to the ledger made by smart contracts within a block, miners execute each transaction in sequential order and write the corresponding changes to the block. If a miner succeeds in appending their block to the chain, validators re-execute the same code, in order to verify that the state changes written by the miner are correct. Platforms such as Ethereum implement fees proportional to the computational steps required to execute smart contracts, called "gas." By executing the smart contracts within their blocks concurrently, miners and validators could perform their work faster, increasing their own profit as well as the potential scalability of the shared ledger. However, this makes it difficult for validators to verify the correctness of the proposed block. A block may be composed of many smart contract invocations, each making multiples accesses to their own state variables or even the state variables of other smart contracts. As a result, different thread inter-

leavings can have an unpredictable effect on the state change computed from a block. In order to guarantee that a block is invalid, validators would need to check every possible interleaving to see if it could produce the state change proposed by the miner of the block. This intractable computation fully negates the performance gain for the miner in executing the block concurrently.

Existing works have proposed solutions for executing smart contracts in parallel. Dickerson et al. [11] propose an approach based on "Transactional Boosting" [17], in which each miner speculatively executes smart contracts in parallel by acquiring a set of locks corresponding to the desired resources. The parallel execution is distributed as a fork-join schedule [3] to validators so that the execution's results can be verified.

In this chapter, we propose a lock-free methodology for concurrent smart contract transactions which utilizes descriptor objects to synchronize thread access to shared smart contract state variables, as well as to form a concurrent history which can be checked by validators. Our approach is based on "Lock-Free Transactional Transformations" [50]. An overview of this approach is given by figure 5.1. Smart contract code is modified to contain a descriptor pointer for all state variables. Whenever a descriptor is replaced using CAS, a pointer is included in the new descriptor, which references to the old one. This naturally produces a directed graph G in which any two transactions that modify the same state variables will have a total ordering. If a block of transactions is re-executed concurrently, obeying the orderings given by G, the final state of the computation is deterministic. This graph G is distributed along with the block, similar to [11]. Afterward, validators can use graph G to deterministically re-execute the block such that their execution matches the original.

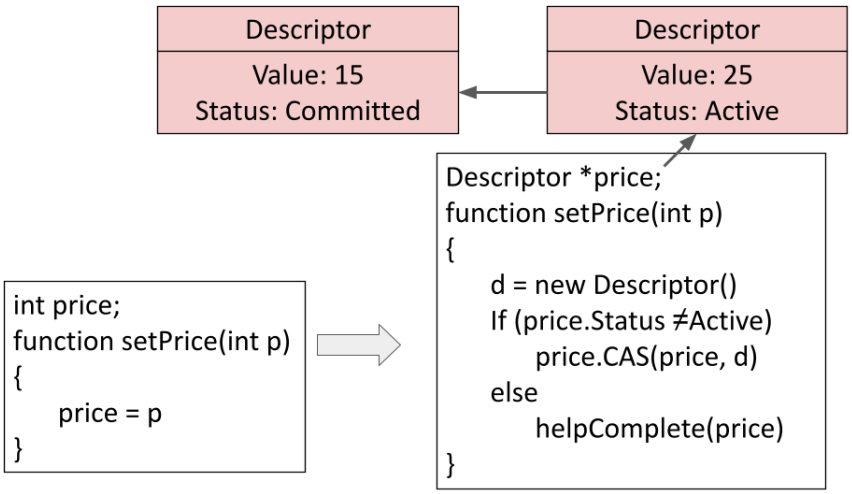

Figure 5.1: Descriptors protect state variables, enabling thread synchronization and naturally capturing transaction conflicts.

Methodology

In this section, we introduce a lock-free two-phase mechanism for concurrent smart contract execution. In the *Primary Execution* phase, a miner executes a block of smart contracts concurrently using an algorithm based on LFTT, and distributes a graph of descriptors to validators. In the *Validation* phase, the block is re-executed by validators, who obey the conflict ordering given by the descriptor graph.

Primary Execution

Algorithm 8 gives the object definitions in our approach. A *Transaction* is a smart contract invocation which has been submitted to the mempool. The *Func* field contains the method being invoked, and the *Data* field contains any parameters. *TxStatus* represents the three possible states of a transaction during a concurrent execution. The *Desc* object is a descriptor used to indicate the

Algorithm 8 System objects

```
1:  struct Transaction
2:      Data
3:      Func
4:  struct TxStatus
5:      Active
6:      Committed
7:      Aborted
8:  struct Desc
9:      Transaction* t
10:     TxStatus status
11:     Set<Desc *> prevs
12:     T returnValues[]
13:     Operation ops[]
14: struct OpInfo
15:     Desc* desc
16:     T value
17:     T prevValue
18:     OpInfo * prev
```

status of a transaction that is being executed concurrently. It contains a *status* field, which can be updated by threads to commit or abort transactions in a single atomic step. Additionally, it contains a concurrent map of pointers to descriptors, *prevs*, representing non-commuting transactions that committed immediately prior to *t*. We apply the approach of DTT [22] by adding a *returnValues* field to each descriptor, which threads can refer to while helping a transaction complete to avoid repeating method calls already completed by other threads. The *OpInfo* class represents a single read/write taking place as part of a transaction. It contains an abstract type *value* which represents the value of a state variable, as well as the previous value *prevValue*. These fields are used to logically interpret the status of a state variable depending on if it commits or aborts. *OpInfo* objects are accessed via pointers, which are added to the smart contract source code. Threads update these pointers using Compare-And-Swap, ensuring that if two threads attempt to access a state variable at the same time, only one will succeed. Additionally, whenever an *OpInfo* pointer is updated using CAS, we include the previous *OpInfo* object in the *prev* field, signifying a happens-before relation

between the two conflicting operations.

Algorithm 9 Pseudocode for a concurrent vending machine smart contract

```
1:  OpInfo *tokens
2:
3:  function DISPENSETOKENS(uint amount, Desc desc)
4:      val = CallOp(desc, tokens, "GetAndIncrement", -amount)
5:      if val ≥ 0 then
6:          return true
7:      else
8:          return false
9:
10: function GETTOKENS(Desc desc)
11:     val = CallOp(desc, tokens, "Get")
12:     return true
```

In order to satisfy strict serializability, each data structure operation must be linearizable. For this reason, we replace any updates within a smart contract with a CAS-based loop. Additionally, since the concurrent execution must be re-executed deterministically by validators, we use each *OpInfo*'s *prev* field to record transaction conflicts as they occur.

Algorithm 9 provides a simple modified smart contract as an example. The smart contract contains a quantity of tokens represented by the state variable *tokens*. The method *DispenseTokens* subtracts a given value from the tokens held by the smart contract. To facilitate concurrent updates, *tokens* is declared as a pointer to an OpInfo object, which contains an abstract type T, representing the value of the state variable. Instead of directly modifying the state variable, we use the library method *CallOp*, passing in the operation type, as well as the amount of tokens to subtract from the state variable. Afterwards, we check if the resulting value has gone below 0. If so, we return *false*, aborting the transaction and logically rolling back any work made by it. Similarly, the *GetTokens* method retrieves the current value of the state variable.

Transaction execution is handled with an approach based on Lock-Free Transactional Transformations [50], and is given by Algorithm 10. This algorithm takes a descriptor, and executes the

Algorithm 10 Driver for transaction execution

```
 1: thread local Stack helpStack
 2:
 3: function EXECUTE(Desc desc)
 4:     helpStack.init()
 5:     ExecuteTransaction(desc)
 6:
 7: function EXECUTETRANSACTION(
    Desc desc)
 8:     if helpStack.contains(desc) then
 9:         desc.status.CAS(ACTIVE, ABORTED)
10:         return
11:     helpStack.push(desc)
12:     ret = desc.t.Func(desc.t.data, desc, localMap)
13:     helpStack.pop()
14:     if ret == true then
15:         desc.status.CAS(ACTIVE, COMMITTED)
16:     else
17:         desc.status.CAS(ACTIVE, ABORTED)
```

corresponding smart contract method call. Threads enter through the *Execute* method, in which a thread local *helpstack* is initialized before *ExecuteTransaction* is called. We inherit the helpstack from LFTT in order to prevent threads from getting stuck in a cyclic dependency while helping transactions complete. If a thread finds a transaction descriptor is already in its help stack on line 10.8, it means there exists a cyclic dependency between some other transaction and *desc*. In order to resolve this and prevent the thread from looping indefinitely, the transaction is aborted. Afterward, the thread reads the method and parameters of the smart contract call from *desc*, and calls the corresponding method. Once the thread has reached the end of the smart contract code path, the status of the transaction is atomically set to committed on line 10.15, signalling to all other threads that the transaction is complete, and any that subsequent conflicting transactions may proceed.

The *CallOp* method is given by algorithm 11. This method takes a state variable pointer *p*, and performs the operation given by *type*, utilizing any arguments given by *args*. On line 11.2, we check if the current transaction is still active by check the status field of its descriptor. Since

Algorithm 11 Call Operation

```
1:  function CALLOP(Desc * desc, OpInfo * p, OpType type, args...)
2:      if desc.status == ABORTED then
3:          return null
4:      opid = helpstack.GetOpId()
5:      if desc.returnValues[opid] exists then
6:          return desc.returnValues[opid]
7:      desc.ops[opid] = new Operation(args)
8:      newInfo = new OpInfo()
9:      newInfo.desc = desc
10:     newInfo.opid = opid
11:     while true do
12:         ret, val = UpdateInfo(p, newInfo, type, args)
13:         if ret == SUCCESS then
14:             break
15:         if ret == FAIL then
16:             return null
17:     desc.returnValues[opid] = val
18:     helpstack.NextOp()
19:     return val
```

threads may work on transactions concurrently, it is possible that another thread completed the transaction, in which case the current thread stops its execution. The id of the current operation is retrieved from the helpstack on line 11.4. This enables the thread to check if there is already a *returnValue* for this operation in the descriptor. If so, we simply return the returnValue stored in the descriptor, avoiding the need to repeat any work completed by another thread. Afterwards, a new OpInfo object is initialized, and passed into the *UpdateInfo* method. The *UpdateInfo* method attempts to update *p* to contain *newInfo*. If successful, the while loop can be broken, otherwise, the loop is retried. Upon success, the new value of the state variable is written to the descriptor's *returnValues* field on line 11.17. Additionally, the helpstack is updated to increment the opid of the current transaction.

Algorithm 12 gives the CAS-based update pattern similar to that of LFTT. First, the current value of *p* is atomically dereferenced, and stored as *oldInfo*. The objective of this method is to atomically

Algorithm 12 CAS-based update

```
1:  function UPDATEINFO(OpInfo * p, OpInfo * newInfo, OpType type, args...)
2:      oldInfo = p.load()
3:      if oldInfo.desc != newInfo.desc then
4:          ExecuteTransaction(oldInfo.desc)
5:      else if oldInfo.opid ≥ newInfo.opid then
6:          return SUCCESS
7:      if oldInfo.desc.status == COMMITTED then
8:          val = oldInfo.value
9:      else
10:         val = oldInfo.prevValue
11:     newInfo.prevValue = val
12:     if type == "Get" then
13:         newInfo.val = val
14:     else if type == "Write" then
15:         newInfo.val = args
16:     else if type == "GetAndIncrement" then
17:         newInfo.val = val + args
18:     if desc.status != ACTIVE then
19:         return FAIL
20:     newInfo.prev = oldInfo
21:     if p.CAS(oldInfo, newInfo) then
22:         return SUCCESS, newInfo.val
23:     else
24:         return RETRY
```

swap the value of *p* from *oldInfo* to *newInfo*. If *oldInfo* references a descriptor that is different to that of *newInfo*, the thread will first help that transaction complete by calling *ExecuteTransaction* on line 12.4. Otherwise, if another thread has already completed the operation, the rest of the operation is skipped (line 12.6). The current value of the state variable is logically interpreted based on the status of the descriptor on line 12.7. If the previous operation committed, we read the *value* field. If the transaction did not commit, we instead read the *prevValue* field, effectively rolling back the changes made by the aborted transaction. The new value of the state variable is computed based on the operation being performed, before CAS is used to update *p* on line 12.21. If the CAS succeeds, it means no other threads attempted to update *p* after its value was read on line

12.2. If another thread did change *p* in that time, the CAS operation will fail, returning a RETRY code. Since *oldInfo* represents the previous operation that modified *p*, we must capture their relative execution order so that the concurrent execution can later be validated deterministically. This is done by storing a reference to *oldInfo* within *newInfo* on line 12.20.

Validation

During the validation phase, all validators on the network re-execute each transaction within the proposed block and verify that the final state proposed by the miner is accurate. In our approach, validators execute the block concurrently using the same lock-free algorithm as in the primary phase, treating the descriptor graph as a fork-join schedule.

Algorithm 13 Graph construction

```
1: function COMPUTEGRAPH
2:     for each state variable s do
3:         currInfo = s.load()
4:         while currInfo != null do
5:             prevInfo = currInfo.prev
6:             if prevInfo != null then
7:                 currInfo.desc.prevs.append(prevInfo.desc)
8:             currInfo = prevInfo
```

Algorithm 13 gives the code for building the graph after all transaction have been executed. The current *OpInfo* object is loaded from each state variable. Since each *OpInfo* object contains a reference to the operation the occurred immediately prior, we finalize the graph by appending each reference to the corresponding descriptor on line 13.7. Afterwards, we set *currInfo* = *prevInfo* and repeat until *currInfo* is *null*. This process iterates over each state change that occurred during execution, requiring only $O(n * m)$ time, where n is the size of the block, and m is the number of operations performed per transaction.

Algorithm 14 Block validation

```
 1: function VALIDATEBLOCK
 2:     leafs ← leafs nodes ∈ ComputeGraph()
 3:     for desc in leafs do
 4:         fork → Validate(desc)
 5:     join all forks
 6:     if final state matches block then
 7:         return valid
 8:
 9: function VALIDATE(Desc *desc)
10:     for prev in desc.prevs
11:         fork → Validate(prev)
12:     join all forks
13:     ExecuteTransaction(desc)
```

The *ValidateBlock* method takes a graph of descriptors given by *ComputeGraph*, and executes each transaction descriptor with respect to its conflict ordering during the primary execution phase. To begin, a thread is spawned for each leaf node in the graph, which call *Validate* on line 14.4. In the *Validate* method, the block is re-executed deterministically using a fork-join approach. On line 14.10, the thread loops through all predecessors of *desc*, containing the transactions that immediately preceded *desc* in the history represented by the descriptor graph. The thread forks for each sub-task by calling *ExecuteTransaction*, passing in the current descriptor. The join operation on line 14.12 ensures that each predecessor of *desc* executes fully before *desc*. After *desc* is executed, the thread compares the descriptor produced by the call to *ExecuteTransaction* to the descriptors produced during the execution phase on line 14.6. If the descriptors differ, then the block fails its validation. If the descriptors match those of the execution phase, then execution proceeds until all transaction have been executed. If none of the generated descriptors differ from the original execution, then the proposed state of the block given by the miner matches the state given by executing the block of transactions concurrently according to the ordering given by the descriptor graph.

Correctness

Our proposed algorithm is designed for the correctness condition strict serializability. The definition of strict serializability is given by Herlihy and Koskien [17]. Our work follow from the proofs of Zhang et al. [50] that LFTT is strictly serializable.

Rule 1. ***Linearizability*** *For any history h, two concurrent invocations I and I' must be equivalent to either the history h · I · R · I' · R' or the history h · I' · R' · I · R*

Rule 2. ***Commutativity Isolation:*** *For any non-commutative method calls $I_1, R_1 \in T_1$ and $I_2, R_2 \in T_2$, either T_1 commits or aborts before any additional method calls in T_2 are invoked, or vice-versa.*

A data structure is strictly serializable if each operation on that data structure is linearizable [19], and each transaction satisfies commutativity isolation. To prove the linearizability of our approach, we identify the linearization points. To prove that transactions in our approach satisfy commutativity isolation, we examine possible transaction conflicts in the code path of our algorithm.

Lemma 1. *The UpdateInfo method is linearizable*

Proof. The linearization points for a variable updated in the style of algorithm 9 occurs on line 12.21, when the CAS operation is executed. It is at this point that the changes made by a thread to some state variable s become visible to all threads. If the operation fails, it means that another thread has succeeded their own CAS operation, and the loop is retried. □

Lemma 2. *Transactions executed by ExecuteTransaction satisfy commutativity isolation*

Proof. Two smart contract method invocation commute if they read or write to a disjoint set of state variables. Let T_1 be an active transaction that has successfully updated the descriptor pointer for a state variable s_1 by executing the CAS on line 12.21. If a transaction T_2 attempts to update s_1, it will read the status of the descriptor placed there by T_1 at line 12.4, and help T_1 complete. In this case, T_1 clearly commits before T_2 is able to access s_1.

If T_1 and T_2 were to arrive at line 12.21 at the same time, having read the same value for *oldInfo*, one will succeed the CAS operation, and the other will fail. On the next loop, the failing thread will, if necessary, help the succeeding transaction complete before proceeding. In this case, the thread that succeeds the CAS operation will commit its transaction before the failing thread is able to update the descriptor. □

Theorem 1. *The proposed concurrent smart contracts algorithm is strictly serializable*

Following from the conclusions of Herlihy and Koskien [17], given Lemma 1 and 2, our proposed algorithm is strictly serializable.

Progress Guarantee

Our approach provides a guarantee of Lock-Free progress. Lock-Freedom guarantees that for any given execution, at least one thread will make progress in a finite amount of steps. To prove this, we analyze the unbounded loop CAS-based loop pattern given by algorithm 11. The while-loop during any state variable read/write terminates when the call to *UpdateInfo* returns *true*. For any

active transaction, *UpdateInfo* returns true if the CAS on line 12.21 succeeds, terminating the loop. If the CAS fails, it means another thread has succeeded their own operation, therefore at least one thread is guaranteed to make progress. In the case that a thread must help a pending operation complete, the maximum number of recursive help calls is equal to the maximum number of active transactions, which is equal to the number of threads *i*. In the worst case, $i - 1$ all help to complete the transactions started by thread i. In this case, all threads will execute the CAS on line 12.21, and at least one will make progress.

Descriptor Graph

In this section, we prove that the references placed in each transaction descriptor during transaction execution represent a strictly serializable history of that execution.

LFTT detects conflicts semantically whenever transactions attempt to modify the same memory location using CAS. In the *UpdateInfo* method, the current state of a memory location *p* is atomically dereferenced on line 12.2. If the CAS operation on line 12.21 succeeds, it means that no updates occurred at *p* after *oldInfo* was read. Thus, the operation described by *oldInfo* reaches its linearization points before the operation described by *newInfo*, with no operations occurring in between. This ordering is captured by including a reference to *oldInfo* within *newInfo* on line 12.20. This produces a total ordering between each pair of non-commuting operations. Since commuting operations can be executed in any order without affecting the final state, the resulting descriptor graph can be executed deterministically to verify the validity of the Primary Execution.

Experimental Evaluation

We benchmark our algorithm by implementing a *Vending Machine* smart contract. In this contract, threads can execute *Vend(i)*, which updates a state variable at array index *i*. For our experiments, *Vend* may be invoked multiple times in a single transaction, increasing the number of state variables accesses per transaction.

We perform experiments to analyze the scalability of our approach in terms of transaction throughput. We generate a block of 10^6 *Vend* calls, containing N state variable accesses. This very large block size allows threads to perform a large number of concurrent transactions, ensuring that threads have an appropriate amount of time to execute concurrently. Indices of operations are selected in a uniform random manner. We hold the number of user keys constant at 10,000 to generate conflict between operations. In many implementations, smart contract methods may make calls to other smart contracts, potentially accessing a large number of state variables in a single transaction. As such, we repeat our experiment for each $N \in 4, 8, 16$, where N is the number of state variable accesses per transaction. We execute our benchmarks on an Intel i7-12700k processor with 8 cores. We execute each algorithm using 1 to 8 threads, taking the average execution time across 10 runs. Timing begins when threads are spawned and begin execution transactions. Timing completes when all transactions have been executed. We measure the throughput in op/ms, and plot each algorithm's throughput against the number of threads.

In order to compare our approach, we implement the transactional boosting approach of Dickerson et al. [11]. For transactional boosting, we pre-allocate locks for each state variable in the experiment. Additionally, we implement an undo log for rolling back transactions upon abort. [1].

Figure 5.2 gives the results of our throughput experiments. We denote our approach "LFTT," and

[1]The source code for our experiments is available at: https://github.com/ZacharyPainter/ConcurrentSmartContracts/tree/main

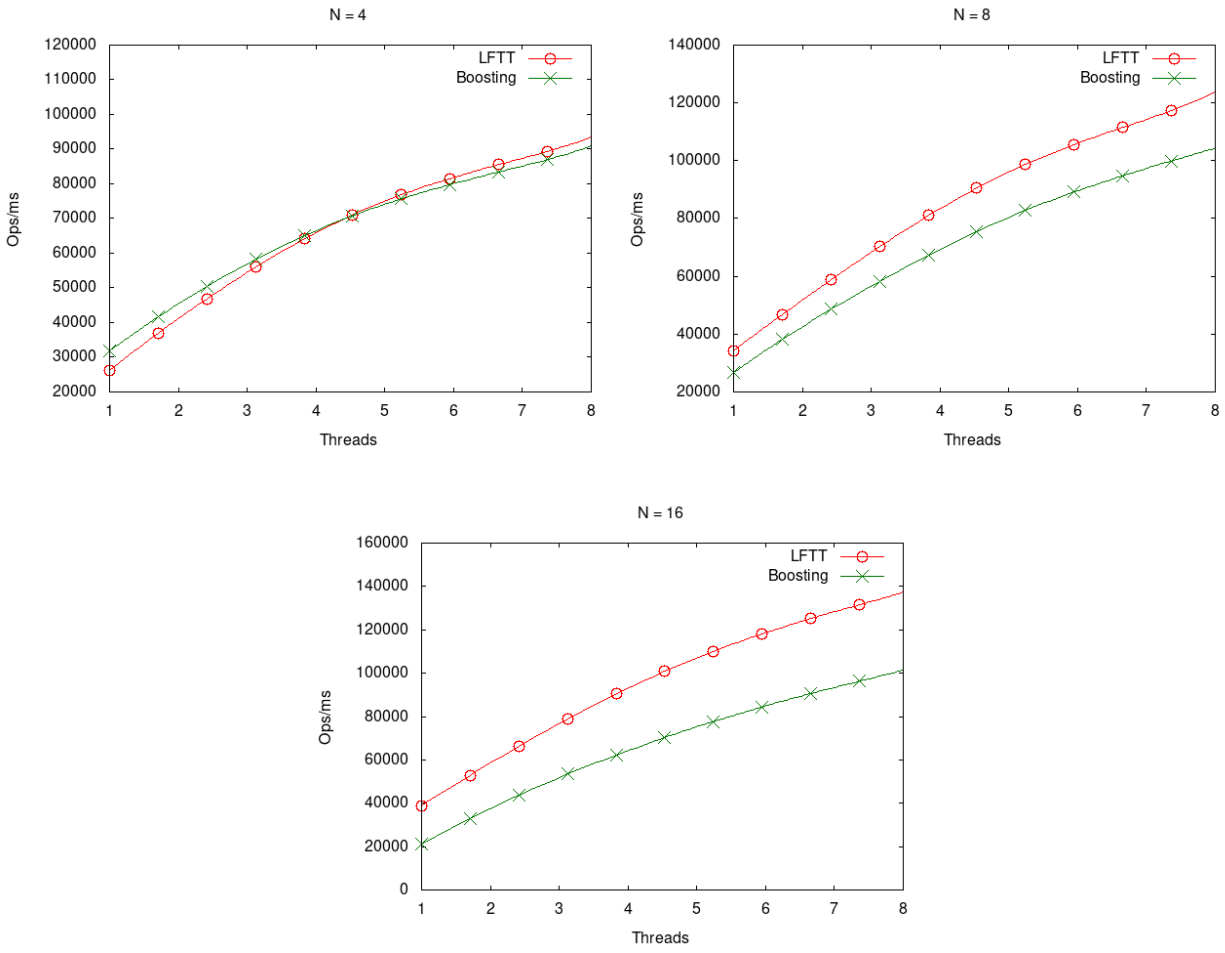

Figure 5.2: Scalability comparison.

transaction boosting approach "Boosting." We observe that as the value of N increases, the cost of transaction rollbacks increases for boosting, causing a decrease in performance relative to LFTT. LFTT outperforms boosting by 20% across each thread count when $N = 8$, and as much as 28% when $N = 16$. This is due to the thread helping scheme preventing transactions from aborting in LFTT except in cases where a cyclic dependency occurs. In boosting, aborted transactions must be physically rolled back, negating any progress made during the transaction in addition to delaying the start of a new transaction.

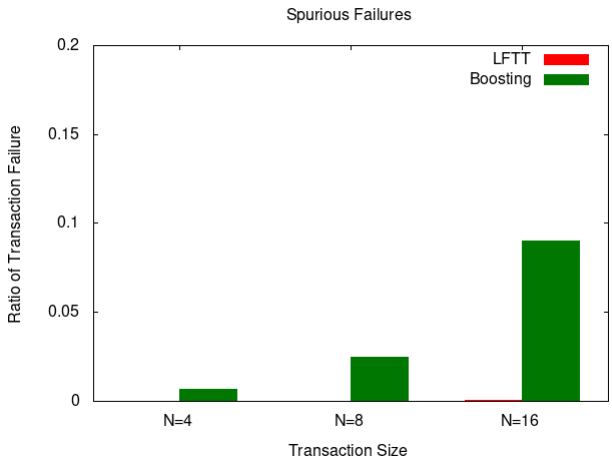

Figure 5.3: Failed transactions comparison.

Figure 5.3 compares the ratio of aborted transactions generated by each algorithm. As transaction size increases, and therefore the likelihood of transaction conflicts, the ratio of spurious aborts in boosting increases up to almost 10% at the largest transaction size (N=16). In comparison, LFTT aborts only .05% of transactions. This is because LFTT only aborts transactions in the case of a cyclic dependency, whereas boosting aborts transactions whenever a conflict is detected. In this way, LFTT dramatically reduces the number of transactions that must be re-executed without coverage by smart contract execution fees.

Chapter Summary

In this chapter, we described a lock-free algorithm for concurrent smart contracts. Our approach uses descriptor objects to synchronize thread access to smart contract state variables, as well as to build a fork-join schedule without the use of locks. Validators utilize the fork-join schedule to verify the correctness of a proposed block using the same lock-free algorithm.

CHAPTER 6: CONCLUSION

In this dissertation, we described several approaches for solving concurrency problems in decentralized networks by adapting known SMP techniques.

In chapter 3, we presented Hash-Mark-Set (HMS). HMS provides blockchain clients with a READ-UNCOMMITTED view of blockchain state variables. Additionally, HMS provides clients with information about transaction dependencies, allowing miners the option to minimize the number of failed transactions per block by choosing a transaction order that respects the dependencies given by HMS. We evaluate our approach in cases where state variables change frequently between blocks, with read to write ratios varying between 1:1 and 20:1. In these tests, our approach improves transaction efficiency by as much as 16x when semantic mining is used, and by an average of 5x when no miner involvement is used.

In chapter 4, we presented Proof of Descriptor (PoD). PoD is a consensus mechanism designed to support concurrent updates to the ledger by clients on the network. This differs from existing consensus mechanisms, which designate a single "leader" to propose the next block. Our approach adapts the use of descriptor objects from LFTT, enabling clients to produce a history of transactions that is strictly serializable, and therefore, equivalent to some history of transactions where each transaction is executed sequentially. In experiments, we demonstrate our approach is capable of outperforming Solana, the fastest sequential blockchain, due to the possibility of concurrent ledger updates.

In chapter 5, we presented an algorithm for lock free execution of smart contracts. This approach is designed for existing blockchains, where it may be infeasible to replace the consensus mechanism with that of Chapter 4. This approach also adapts the descriptor-based execution of LFTT to enable all transactions with a block to be executed concurrently. The descriptor objects created during this

process are re-purposed to form a graph representing all transaction conflicts that occurred during execution. This graph is used by validators to re-execute the block deterministically, ensuring that they achieve the same computational result as the miner. Our approach outperforms related approaches due to its better handling of contention, and efficient construction of the dependency graph.

LIST OF REFERENCES

[1] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77. ACM, 2018.

[2] L M Bach, B Mihaljevic, and M Zagar. Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550, May 2018.

[3] Robert D Blumofe, CF Joerg, BC Kuszmaul, CE Leiserson, KH Randall, and Y Zhou. An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP*, page 207.

[4] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[5] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[6] Vincent Chu. Ethereum private net. `https://github.com/vincentchu/eth-private-net`, September 2017. Accessed: 2018-7-1.

[7] Consolidated Tape Association. US consolidated tape data. `https://www.ctaplan.com/publicdocs/ctaplan/notifications/trader-update/Q2_2019_CTA_SIP-Subscribers_Metrics_Report.pdf`, July 2019.

[8] Victor Cook, Zachary Painter, Peterson Christina, and Damian Dechev. Read-uncommitted transactions for smart contract performance. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1960–1970, 2019.

[9] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[10] Primavera De Filippi, Morshed Mannan, and Wessel Reijers. Blockchain as a confidence machine: The problem of trust & challenges of governance. *Technol. Soc.*, 62:101284, August 2020.

[11] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 303–312, New York, NY, USA, July 2017. Association for Computing Machinery.

[12] David Easley, Maureen O'Hara, and Soumya Basu. From mining to markets: The evolution of bitcoin transaction fees. May 2018.

[13] Ethereum. Eip-2982: Serenity phase 0. `https://eips.ethereum.org/EIPS/eip-2982`, 2020.

[14] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.

[15] Agata Fronczak, Piotr Fronczak, and Janusz A. Hołyst. Average path length in random networks. *Phys. Rev. E*, 70:056110, Nov 2004.

[16] Valentina Gatteschi, Fabrizio Lamberti, Claudio Demartini, Chiara Pranteda, and Víctor Santamaría. Blockchain and smart contracts for insurance: Is the technology mature enough? *Future Internet*, 10(2):20, February 2018.

[17] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, 2008.

[18] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[19] Maurice Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[20] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

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

[22] Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. Wait-free dynamic transactions for linked data structures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 41–50, 2019.

[23] Guillermo Jesús Larios-Hernández. Blockchain entrepreneurship opportunity in the practices of the unbanked. *Bus. Horiz.*, 60(6):865–874, November 2017.

[24] Ton Chanh Le, Lei Xu, Lin Chen, and Weidong Shi. Proving conditional termination for smart contracts. In *Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts*, pages 57–59. ACM, 2018.

[25] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[26] Gianmaria Del Monte, Gianmaria Del Monte, Diego Pennino, and Maurizio Pizzonia. Scaling blockchains without giving up decentralization and security, 2020.

[27] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13. Washington, DC, 1975.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

[29] Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, 2007.

[30] Zachary Painter, Victor Cook, Christina Peterson, and Damian Dechev. Descriptor based consensus for blockchain transactions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, DEBS '21, pages 114–125, New York, NY, USA, June 2021. Association for Computing Machinery.

[31] Zachary Painter and Damian Dechev. Lock free concurrent smart contracts. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, Forthcoming.

[32] Zachary Painter, Pradeep Kumar Gayam, Victor Cook, and Damian Dechev. Parallel hash-mark-set on the ethereum blockchain. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–5. IEEE, 2020.

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

[34] M A Patil and P T Karule. Design and implementation of keccak hash function for cryptography. In *2015 International Conference on Communications and Signal Processing (ICCSP)*, pages 0875–0878, April 2015.

[35] Tom Popomaronis. Prime day gives amazon over 600 reasons per second to celebrate. `https://www.inc.com/tom-popomaronis/amazon-just-eclipsed-records-selling-over-600-items-per-second.html`, July 2016. Accessed: 2020-1-13.

[36] Dian Rachmawati, JT Tarigan, and ABC Ginting. A comparative study of message digest 5 (md5) and sha256 algorithm. In *Journal of Physics: Conference Series*, volume 978, page 012116. IOP Publishing, 2018.

[37] M Staples, S Chen, S Falamaki, A Ponomarev, P Rimba, A B Tran, I Weber, X Xu, and L Zhu. Risks and opportunities for systems using blockchain and smart contracts. *Data61 (CSIRO), May*, 2017.

[38] Melanie Swan. *Blockchain: Blueprint for a New Economy*. "O'Reilly Media, Inc.", January 2015.

[39] Marcus Holst Swende. Blockchain frontrunning. `http://swende.se/blog/Frontrunning.html`, July 2017. Accessed: 2019-1-9.

[40] Alex Tapscott and Don Tapscott. How blockchain is changing finance. *Harv. Bus. Rev.*, 1(9):2–5, 2017.

[41] Elie Track, Nancy Forbes, and George Strawn. The end of moore's law. *Comput. Sci. Eng.*, 19(2):4–6, March 2017.

[42] P Treleaven, R Gendal Brown, and D Yang. Blockchain technology in finance. *Computer*, 50(9):14–17, 2017.

[43] VISA Corporation. VISA fact sheet. `https://usa.visa.com/dam/VCOM/global/about-visa/documents/visa-fact-sheet-july-2019.pdf`, July 2019.

[44] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-Work vs. BFT replication. In *Open Problems in Network Security*, pages 112–125. Springer International Publishing, 2016.

[45] Y Wanjun and W Yuan. Research on network trading system using blockchain technology. In *2018 International Conference on Intelligent Informatics and Biomedical Sciences (ICI-IBMS)*, volume 3, pages 93–97, October 2018.

[46] Ethereum Wiki. On sharding blockchains faqs, 2021. https://eth.wiki/sharding/Sharding-FAQs.

[47] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain. https://solana.com/solana-whitepaper.pdf.

[48] Anatoly Yakovenko. Proof of history: A clock for blockchain, 2021. https://medium.com/solana-labs/proof-of-history-a-clock-for-blockchain-cf47a61a9274.

[49] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.

[50] Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. Lock-free transactional transformation for linked data structures. *ACM Transactions on Parallel Computing (TOPC)*, 5(1):1–37, 2018.