2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

Read-Uncommitted Transactions for Smart Contract Performance

Victor Cook*, Zachary Painter†, Christina Peterson‡, Damian Dechev §

Computer Science Department, *University of Central Florida*
Orlando, USA
*victor.cook@knights.ucf.edu, †zacharypainter@knights.ucf.edu,
‡clp8199@knights.ucf.edu, §dechev@cs.ucf.edu

***Abstract*—Smart contract transactions demonstrate issues of performance and correctness that application programmers must work around. Although the blockchain consensus mechanism approaches ACID compliance, use cases that rely on frequent state changes are impractical due to the block publishing interval of $O(10^1)$ seconds. The effective isolation level is READ-COMMITTED, only revealing state transitions at the end of the block interval. Values read may be stale and not match program order, causing many transactions to fail when a block is committed. This paper perceives the blockchain as a transactional data structure, using this analogy in the development of a new algorithm, *Hash-Mark-Set* (HMS), that improves transaction throughput by providing a READ-UNCOMMITTED view of state variables. HMS creates a directed acyclic graph (DAG) from the pending transaction pool. The transaction order derived from the DAG is used to provide a READ-UNCOMMITTED view of the data for new transactions, which enter the DAG as they are received. An implementation of HMS is provided, interoperable with Ethereum and ready for use in smart contracts. Over a wide range of transaction mixes, HMS is demonstrated to improve throughput. A side product of the implementation is a new technique, *Runtime Argument Augmentation* (RAA), that allows smart contracts to communicate with external data services before submitting a transaction. RAA has use cases beyond HMS and can serve as a lightweight replacement for blockchain oracles.**

***Index Terms*—Blockchain, Smart contracts, Concurrent algorithms, Transaction throughput**

I. INTRODUCTION

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 [1], [2] 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 paper, we present *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.

An interoperable implementation of the solution is provided and demonstrated on the Ethereum blockchain. State throughput, defined in Section III-A as throughput of *successful* blockchain transactions, increases by a factor of five across a range of transaction mixes. By adding the cooperation of

This work was funded by the National Science Foundation (NSF) under Grant Numbers 1717515 and 1740095.

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00194

blockchain miners, state throughput increases by an order of magnitude.

We make the following contributions:

1) Present *Hash-Mark-Set* (HMS), an algorithm organizing a pool of pending transactions that share state variables.
2) Introduce *Runtime Argument Augmentation* (RAA), a novel technique for smart contracts to communicate with external data services.
3) Demonstrate improvements to latency and state throughput when HMS provides a READ-UNCOMMITTED view to smart contract clients.
4) Provide an interoperable implementation for Ethereum: https://github.com/area67/sereth

II. BACKGROUND

A few issues specific to the blockchain are discussed in this section.

A. Blockchain Transactions

A *blockchain* is a distributed ledger maintained by one or more peers following a communication protocol and agreeing on a consensus mechanism. The ledger is written in chunks called *blocks* that are linked in a practically unforgeable cryptographic *chain*, replicated among many peers to avoid dependence upon a single entity. State variables that are recorded on the blockchain are called *storage* variables in Ethereum. Bitcoin was the first blockchain [3], providing transactions on a store and exchange of value, *i.e.* a currency. A transaction is a concurrent method call that if successful, changes the state of the ledger. A block may contain from zero up to a finite number of transactions, typically $O(10^1)$ to $O(10^3)$. Well known database transaction models such as ACID and BASE are applicable to the blockchain [4], motivating our use of isolation levels READ-UNCOMMITTED and READ-COMMITTED.

B. Concurrent Smart Contracts

Going beyond exchange of value, later blockchains added the ability to program arbitrary instruction sequences in a transaction. Their programming languages are Turing complete [5] and their programs are called *smart contracts* [6]. Concurrency is framed in the words of Sergey and Hobor, "Accounts using smart contracts in a blockchain are like threads using concurrent objects in shared memory" [7]. Herlihy endorsed this line of reasoning in a keynote address [8], exhorting concurrency researchers to "civilize" the blockchain.

Invoking a smart contract function that *may* change ledger state creates a transaction and sends it to the network. It should be noted that some smart contract functions, designated *pure* or *view*, cannot change ledger state and they do not create transactions. The unprocessed transaction pool of pending transactions is referred to as the *TxPool*. The network of peers is a concurrent system and it follows that its incoming transactions, found in the TxPool, is a concurrent history. The *real time ordering* of a concurrent history is a total ordering over the transactions in a concurrent history such that transaction T_1 is ordered before transaction T_2 if T_1 is received by the TxPool before T_2.

C. Miner Privilege

On Ethereum, Bitcoin and many other blockchains, the inclusion and sequencing of transactions in a block does not follow real time order, rather transactions are arranged in a total order that is arbitrary and subject to the same economic incentives that drive blockchain progress [9]. This is called the *block order*. Special peers, called *miners* have the privilege of deciding what goes into a block and in what order. Each transaction is isolated and a miner generally has no way of knowing if one may depend upon another, so the rules for selecting transactions are flexible. Miners generally favor transactions with higher fees, but they may favor some peers, including themselves. They may use altruistic criteria such as including only small transactions or those from peers with low bandwidth.

The discretion given to miners in the protocol works as if the scheduler of a CPU could favor particular threads. Ethereum miners read the TxPool grouped by peer addresses (aka threads) with transactions ordered by a counter called a *nonce*. Miners may favor an address and include its transactions before another peer without regard for the real time order in which they were received. Miners may refuse to include any transactions sent from particular addresses. But a miner may not commit a transaction from a given address to a block out of nonce order. This means that blockchain transactions from the same address are executed in the order they are sent, while the order of transactions from different addresses is not defined. Since a blockchain transaction is a concurrent method, we can describe this behavior as being equivalent to *sequential consistency*, a correctness property such that history of methods is equivalent to a legal sequential history, and all methods take effect in program order [10].

The TxPool is shared by peers on the network, including miners. Intuitively, if communication were instantaneous, all peers would see the same TxPool, and the order in which the transactions were received would match their real time ordering, i.e. the order in which they were sent. Miner privilege would still allow the transactions to be placed in a block in an order different from the real time order. The outcome of our READ-UNCOMMMITTED view of state is subject to network synchronization and miner privilege. Information about the TxPool is not available to the smart contract as it submits transactions.

D. Block Publishing and Validation

Blocks of selected transactions are committed all at once in a super transaction called *block publishing*. Transactions are interpreted sequentially within a block according to the block order, using the previous ledger state (block) as the initial context. Changes to storage variables are not visible until they are committed to a block and the block is published. The changes to storage variables during the interval of block publication are called *intra-block* changes. Transactions

within the block are affected by the intra-block changes, but post-publication transactions read the block final value of a storage variable from the previous block, none of the current changes. Once published there is no opportunity to re-order the concurrent methods. These values were read from the previous block, published *block interval* seconds ago. The block interval defines the latency.

Block publishing is effectively a read lock until the next block is committed. Dirty reads are not allowed. In database terms the isolation level of intra-block transactions is READ-COMMITTED. To accept a published block every peer must perform *block validation*, the task of checking that the block is consistent with the state of the network. Transactions committed to a block must be consistent in that they must include the effects of all previous transactions. The process of peers redundantly validating transactions in a block is called transaction *replay*. Block publishing and validation takes a significant amount of time $O(10^1)$ to $O(10^2)$ seconds, creating latency.

Since only the final state of the block is published, intermediate states become invisible without a detailed replay of the transactions in the block, something that a typical smart contract cannot do. The loss of intermediate states during a block update is a consequence of the READ-COMMITTED view of state variables. This low isolation level avoids blocking but may allow a great number of transactions to be rejected later as inconsistent. Transactions that seem valid when submitted are rejected because the values on which they are based were stale. The number of transactions that are rejected impacts state throughput. Where state changes are frequent and there are many transactions in the pool to be interleaved in a block, a large percentage of transactions fail. To say a transaction failed means that it would have violated the consistency of the sequential history of the block in which it is embedded. To keep the sequential history of the block consistent, the transaction is included in the block, but has no effect on the system state. In database terms the transaction was *rolled back*. A principle cause of failure is the high latency imposed on reading changes to persistent storage variables.

E. Blockchain Oracles

A characteristic of the blockchain is that security concerns related to the adversarial distributed environment impose restrictions on information transfer. Unassisted, smart contracts operate in a bubble, allowed to view only public blockchain state variables via getter functions and not allowed to call any outside sources of information. The discussion in Section II-D about peers replaying blocks can explain this. Since all peers must replay and validate the block, they all must see the same state changes. If a contract is using an outside source of information, no matter how reliable, it may change with time or due to corrections or it may become unavailable. This would cause some peers to see a different state than others, and the block could not be validated. The problem can be solved with a smart contract that mediates a secure and verifiable connection to external data feeds [11]. Such a service is also called a *blockchain oracle* [12], [13].

F. Challenging Use Cases

Blockchain performance, measured in terms of transaction throughput and latency, is a limiting factor for many use cases [14]–[17]. Latency and throughput are considered together in this paper 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" [18]. 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* [19].

III. METHODOLOGY

This paper 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 in the example of Section II-F. 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 paper.

A. 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 II-F, 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.

$$\frac{T_{state}}{T_{raw}} = \eta \tag{1}$$

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.

B. 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 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 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.

Listing 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);
        }
    }
}
```

C. 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 [20] 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 1 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 1 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)
```

Algorithm 2 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
```

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 1:8 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 2 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 1:3, we check on line 1:4 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 3 contains the functions which return the most valid series from a list of Sereth transactions.

Line 3:1 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 3:9 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.

D. 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 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

Algorithm 3 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)
```

is a modification to the Ethereum Virtual Machine (EVM) interpreter, written in Golang. Figure 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

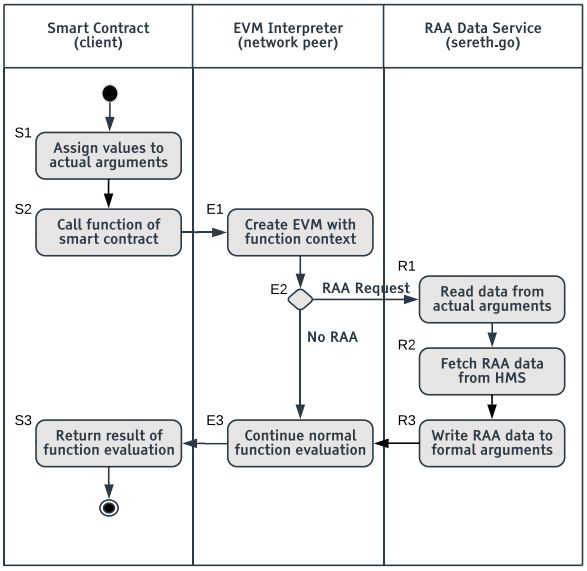

Fig. 1. RAA activity diagram

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.

IV. 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 [10]. 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 [21], [22]. Since the TxPool is a finite list of transactions, Algorithm 2 trivially terminates. Algorithm 1 and Algorithm 3 terminate given that the recursive function DEEPESTBRANCH terminates. We now show in the following lemma that DEEPESTBRANCH terminates.

Lemma 2. DEEPESTBRANCH *presented in Algorithm 3 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 3 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 3. 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 3. □

V. 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 [23]. 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 paper: 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 II-F 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 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.

A. 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 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 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 [24].

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.

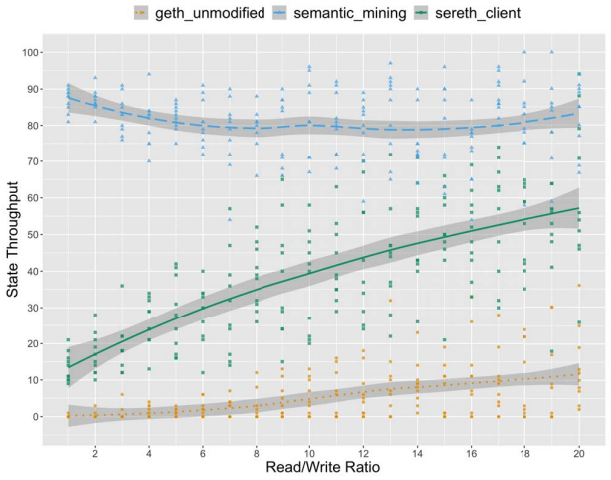

Fig. 2. Transaction efficiency η vs READ-UNCOMMITTED / WRITE ratio

Sets are not plotted. All of the sets succeed because they are sent from the owner of the contract and they do not depend on the previous price. If sets came from different addresses some might have failed, but it is reasonable that the owner of the contract is the only one allowed to set the price.

B. Hash-Mark-Set without miner assistance

The second experimental scenario, labeled as "sereth_client" in Figure 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 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 II-F.

C. 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 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.

VI. RELATED WORK

There are five main approaches to improve blockchain throughput and latency: Reparameterization, sidechains, sharding, leader election and invalid state tagging.

Reparameterization involves tuning the block size and interval to network bandwidth and peer computing power [25]. HMS does not use reparameterization, but could influence tuning trade-offs by decreasing the significance of block interval.

Sidechains [26] increase throughput by creating transaction channel networks such as Lightning [27]. As the name implies they exploit parallelism by running multiple chains, merging them to the main chain as needed to ensure correctness. Sidechains have been implemented at scale on existing blockchains. Recently generalized formally as *state channels* [28], they can provide throughput gains of several orders of magnitude. However the authors note state channels do not solve the latency, or as they call it, "time granularity" problem. The READ-UNCOMMITTED view provided by HMS does solve this for specific state variables.

Sharding [29] increases throughput by isolating segments of the blockchain peer network. Sharding requires changes to consensus protocol but has been accepted by Ethereum [30] with significant progress [31] and a target implementation date in 2020. Like state channels, sharding is inherently parallel and offers performance gains of several orders of magnitude. Sharding is a global solution but would need customization to address state throughput of individual smart contracts as does HMS.

Bitcoin-NG "Next Generation" [18] uses leader election with continuous serialization to modify the consensus protocol in proof of work blockchains such as Bitcoin. Performance gains scaling to the limits of network latency and individual peer processing power are reported. Recent work [32] notes the history of improvement and elaboration on the original proposal. Our solution shares with Bitcoin-NG the concept of continuous serialization to reduce the "long freeze" of latency, but HMS does not require protocol changes to interoperate with current blockchains.

The scope of our review was public blockchains, however a Byzantine Fault Tolerant (BFT) proposal for Hyperledger is relevant because it focused on the bottleneck of transaction signing and ordering in block creation [33]. BFT uses a leader to coordinate block creation achieving transaction rates of over 2000 per second on private blockchains. Unlike Ethereum, Hyperledger tags transactions known to be based on invalid states before they are ordered in a block, so time is not wasted replaying the failed transactions. However the authors do not consider transaction efficiency and the BFT ordering service does not use semantics to reduce failures as HMS does.

Software Transactional Memory (STM) algorithms have also been applied to blockchain throughput. The original ideas in [34] using STM to enable concurrent processing of smart contract methods were continued by [35]. These researchers note the EVM is not parallel and the difficulty of determining transaction dependencies in a block, so in both papers smart contracts were translated into C++ which is supported by the STM library. Simulated miners then interleave and order smart contract methods in STM to create a concurrent execution. Speedups of up to 2x were achieved. Concurrency based throughput gains in which "any sequential execution will do" are different from HMS, which sequences smart contract methods for transaction efficiency and increased state throughput.

A parallel to HMS is found in an earlier STM paper [36] whose language about "publishing" is prescient as it was written before the blockchain was invented. A correctness condition called Selective Strict Serialization (SSS) is introduced, in which some transactions are strictly serialized and others are not, but are marked to the serialized history. In Section IV above we applied sequential consistency as the correctness condition for our HMS algorithm. In our experiments HMS establishes a fixed ordering for the state changes (sets) while allowing the dependent transactions (buys) to be arbitrarily interleaved. Multiple buys can occur in a price interval and are not dependent upon each other. Within the interval any order of buys is valid so they do not require an established ordering constraint. Further work might show that SSS is a correctness condition suitable for HMS.

There is relevant work on improving throughput and latency of concurrent systems by reducing *abort rate*, defined as how many times a transaction is retried before success [37]–[40].

This is different from our *state throughout*, which measures efficiency of blockchain commits that are not repeated. High abort rates due to delayed write visibility, where transaction writes may only be read after commit, is addressed by Faleiro et al. [41] in the proposal of piece-wise visibility (PWV), a deterministic concurrency control protocol designed to enable early write visibility. PWV divides a transaction into a set of sub-transactions which are scheduled to be executed in a serializable order. Each sub-transaction write is made visible as soon as it commits, enabling the original transaction writes to be visible prior to commit time. A DAG is used to order database sub-transactions based on data dependencies. HMS uses a DAG to order blockchain transactions in a sequentially consistent fashion, and the final series of transactions is derived from the deepest branch.

The fundamental difference between PWV and HMS is that PWV enables writes to be visible inside the commit protocol while HMS enables READ-UNCOMMITTED isolation for smart contracts through our proposed RAA technique, described in Section III-D. The PWV commit protocol only provides write visibility after a transaction is submitted to the database system, which limits the potential performance gains in comparison to HMS that provides write visibility to smart contract clients such that the requested data from RAA can be utilized prior to transaction submission.

VII. CONCLUSIONS

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.

ACKNOWLEDGMENT

Thanks to Raul Jordan and Andreas Olofsson for assistance in understanding the intricacies of Geth client software.

REFERENCES

[1] M. Herlihy and E. Koskinen, "Transactional boosting," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPoPP 08*, 2008.

[2] D. Zhang and D. Dechev, "Lock-free transactions without rollbacks for linked data structures," *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures - SPAA 16*, 2016.

[3] S. Nakamoto, "Bitcoin: A Peer-to-Peer electronic cash system," https://bitcoin.org/bitcoin.pdf, Oct. 2008.

[4] S. Tai, J. Eberhardt, and M. Klems, "Not ACID, not BASE, but SALT - a transaction processing perspective on blockchains," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2017, pp. 755–764.

[5] V. Buterin, "Ethereum: The ultimate smart contract and decentralized application platform," http://web.archive.org/web/20131228111141/http://vbuterin.com:80/ethereum.html, Dec. 2013, accessed: 2018-6-30.

[6] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, Sep. 1997.

[7] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Financial Cryptography and Data Security*. Springer International Publishing, 2017, pp. 478–493.

[8] M. Herlihy, "Blockchains and the future of distributed computing," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017, pp. 155–155.

[9] W. Wang, D. T. Hoang, Z. Xiong, D. Niyato, P. Wang, P. Hu, and Y. Wen, "A survey on consensus mechanisms and mining management in blockchain networks," May 2018.

[10] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[11] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 270–282.

[12] X. Xu, C. Pautasso, L. Zhu, V. Gramoli, A. Ponomarev, A. B. Tran, and S. Chen, "The blockchain as a software connector," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. ieeexplore.ieee.org, Apr. 2016, pp. 182–191.

[13] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," in *Financial Cryptography and Data Security*, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Cham: Springer International Publishing, 2017, pp. 494–509.

[14] M. Swan, *Blockchain: Blueprint for a New Economy*. "O'Reilly Media, Inc.", Jan. 2015.

[15] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaría, "Blockchain and smart contracts for insurance: Is the technology mature enough?" *Future Internet*, vol. 10, no. 2, p. 20, Feb. 2018.

[16] 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.

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

[18] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *NSDI*, 2016, pp. 45–59.

[19] M. H. Swende, "Blockchain frontrunning," http://swende.se/blog/Frontrunning.html, Jul. 2017, accessed: 2019-1-9.

[20] 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)*, Apr. 2015, pp. 0875–0878.

[21] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2018, pp. 66–77.

[22] T. C. Le, L. Xu, L. Chen, and W. Shi, "Proving conditional termination for smart contracts," in *Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts*. ACM, 2018, pp. 57–59.

[23] V. Chu, "Ethereum private net," https://github.com/vincentchu/eth-private-net, Sep. 2017, accessed: 2018-7-1.

[24] D. Easley, M. O'Hara, and S. Basu, "From mining to markets: The evolution of bitcoin transaction fees," May 2018.

[25] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains," in *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2016, pp. 106–125.

[26] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and W. Pieter, "Enabling blockchain innovation with pegged sidechains," https://blockstream.com/wp-content/uploads/2014/10/sidechains.pdf, Oct. 2014, accessed: 2018-8-22.

[27] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments,(2016)," *DRAFT Version 0. 5*, vol. 9, 2015.

[28] J. Coleman, L. Horne, and L. Xuanji, "Counterfactual: Generalized state channels," https://l4.ventures/papers/statechannels.pdf, Jun. 2018, accessed: 2018-9-15.

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

[30] V. Buterin, "On sharding blockchains," https://github.com/ethereum/wiki/wiki/Sharding-FAQs, Aug. 2018, accessed: 2018-9-5.

[31] Prysmatic Labs, "Sharding updates," https://medium.com/prysmatic-labs, Sep. 2018, accessed: 2018-9-14.

[32] J. Yin, C. Wang, Z. Zhang, and J. Liu, "Revisiting the incentive mechanism of Bitcoin-NG," in *Information Security and Privacy*, ser. Lecture Notes in Computer Science, W. Susilo and G. Yang, Eds. Cham: Springer International Publishing, 2018, vol. 10946, pp. 706–719.

[33] J. Sousa, A. Bessani, and M. Vukolic, "A byzantine Fault-Tolerant ordering service for the hyperledger fabric blockchain platform," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2018, pp. 51–58.

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

[35] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for concurrent execution of smart contracts," Sep. 2018.

[36] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "Ordering-Based semantics for software transactional memory," in *Principles of Distributed Systems*. Springer Berlin Heidelberg, 2008, pp. 275–294.

[37] Y. Wu and K.-L. Tan, "Scalable In-Memory transaction processing with HTM," in *USENIX Annual Technical Conference*. usenix.org, 2016, pp. 365–377.

[38] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan, "Fast In-Memory transaction processing using RDMA and HTM," *ACM Trans. Comput. Syst.*, vol. 35, no. 1, pp. 3:1–3:37, Jul. 2017.

[39] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang, "BCC: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases," *Proceedings VLDB Endowment*, vol. 9, no. 6, pp. 504–515, Jan. 2016.

[40] N. Cohen, E. Petrank, and J. R. Larus, "Reducing transaction aborts by looking to the future," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: ACM, 2018, pp. 385–386.

[41] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 613–624, 2017.