

Parallel Hash-Mark-Set on the Ethereum Blockchain

1st Zachary Painter

2nd Pradeep Kumar Gayam

3rd Victor Cook

4th Damian Dechev

University of Central Florida

University of Central Florida

University of Central Florida

University of Central Florida

zacharypainter@knights.ucf.edu

pradeep@knights.ucf.edu

victor.cook@knights.ucf.edu

dechev@cs.ucf.edu

Abstract—Popular blockchains such as Bitcoin or Ethereum provide a transaction isolation level of READ-COMMITTED. This provides difficulties when state changes many times per block interval. Hash-Mark-Set (HMS) alleviates this problem by enabling READ-UNCOMMITTED transactions for state variables. However, the current HMS implementation relies on a sequential algorithm and is susceptible to redundant calculations. As modern processors rely more heavily on parallel algorithms to leverage multiple cores for speedup, sequential algorithms see less benefit from hardware improvements. This paper proposes a lock-free HMS to make use of thread-safe techniques and other optimizations to improve the performance of the HMS algorithm and reduce the latency of read-uncommitted state variable accesses. In our experiments, the proposed algorithm experiences an average 6.4x increase in performance up to 128 go-routines, and a maximum 11.1x increase.

I. INTRODUCTION

A blockchain is a decentralized distributed ledger capable of storing state information. Blockchains like Bitcoin [10] or Ethereum [13] append updates to the system as blocks connected by a cryptographically secure chain. Each block contains transactions which modify state variables. Participants in a blockchain use a consensus mechanism to agree on the next block to be added to the sequence. These participants are rewarded for successfully appending their own proposed block to the sequence. Since participants that are able to produce blocks faster are more likely to successfully publish their block to the network, it is beneficial for mining algorithms to efficiently utilize hardware resources.

Lock-freedom [5] is a progress-guarantee for concurrent algorithms that ensures that a system will make progress in a finite number of steps. This property is desirable for concurrent algorithms as it prevents some drawbacks of lock-based counterparts, such as deadlock, while offering effective use of multi-core processors. The design and implementation of lock-free algorithms is challenging, as many possible thread-interleavings must be considered. Linearizability [6] is a correctness condition for concurrent systems. A method is linearizable if its effects occur instantaneously between its invocation and return, and the history produced by calls to the method are equivalent to a valid sequential history.

Peers viewing a blockchain network are only able to view the system's state at the most recent published block. This

This research was supported by the National Science Foundation under NSF CCF 1717515, and NSF OAC 1740095.

transaction isolation level is called READ-COMMITTED [9]. Hash-Mark-Set (HMS) [3] enables a READ-UNCOMMITTED view of storage variables by tagging transactions with an order-defining hash value, and organizing all uncommitted transactions based on this hash value into a directed-acyclic-graph (DAG). The longest traversal of nodes in this DAG constitutes the *series*, or the sequence of transactions that maximizes transaction success rate.

The HMS algorithm must be executed for each new transaction accessing a desired state variable. By creating transactions based on the HMS series instead of the currently published block, HMS improves the *state throughput*, or the ratio of successful transactions, of each block. This increase is even larger with *semantic mining*, when miners construct blocks that strictly obey the ordering of the HMS series. State throughput divided by raw throughput yields the transaction efficiency η (1).

$$\frac{T_{state}}{T_{raw}} = \eta \quad (1)$$

The existing HMS implementation relies on a sequential algorithm, but doing so can introduce a bottleneck when processing a large volume of transactions. In this case, the staleness of reads from HMS will increase, since a lengthy calculation must occur before the state of the variable is known. In this time, it is possible for the state of the variable to change again. This can negatively affect state-throughput as well as it inhibits the volume of HMS transactions that can be processed in a given time frame. Additionally, in the case of semantic mining, miners may be slowed by the runtime of the algorithm. These drawbacks can be minimized by storing a complete history of HMS transactions, and by utilizing fast, highly scalable algorithms for use with modern multiprocessors.

In this paper, we propose a lock-free implementation of HMS that demonstrates an average case 6.4x increase in operation throughput over the existing implementation. We provide an implementation within Geth [4], as well as a test harness to demonstrate performance and state throughput. As demonstrated in section IV, the implementation is linearizable and lock-free.

Our implementation evenly divides the pending transaction pool among threads and locates HMS transactions that have not yet been processed. A node structure is created from each transaction and is placed in a shared array. We connect nodes that share a parent-child relation using atomic COMPAREANDSWAP [8] on the parent's *next* field. Each node's

parent is located in either the shared array or the DAG, which enabled new nodes to be appended in any order with minimal contention. To eliminate the need for expensive searches to find the deepest node in the DAG, we implement a *tail* pointer. When a node is successfully inserted, we calculate its depth based on the depth of its parent. If this new node has a larger depth than the current tail, we use COMPAREANDSWAP to redirect the tail pointer to the new node.

This paper makes the following contributions:

- 1) We improve the performance of HMS by up to 10x when processing large transaction pools.
- 2) We provide an integrated solution for lock-free HMS within Geth.¹

II. BACKGROUND

A. Miners and Transactions

Blocks are appended to the blockchain by miners after resolving a consensus algorithm (PoW [10], PBFT [2], PoA [1], PoS [7]). Each block consists of a sequence of transactions which, when executed in order, modify the state of the blockchain. The transactions which are included in a block, as well as their order, is chosen by the miner. Smart Contracts [12] allow the execution of arbitrary code as part of blockchain transactions. Conflict between transactions modifying the same state variables may result in transactions failure. Failed transactions still occupy space in a block, but ultimately have no effect on state variables.

B. Hash-Mark-Set

Hash Mark Set appends a *mark* field to each transaction acting on a smart contract state variable. A transaction's mark is used to determine its execution order relative to other transactions modifying the same state variable. Since blockchain miners could mine transactions in almost any order, the transaction history created by HMS is not necessarily the same history that will be published in a block. However, the algorithm is demonstrated to be effective at increasing the likelihood of successful transactions even when miners are unaware of the DAG created by HMS.

The HMS algorithm can be broken into 3 steps.

- 1) Read the pending *TxPool*, filter all irrelevant transactions
- 2) Create a Directed Acyclic Graph from all transactions in (1)
- 3) Find the longest traversal of graph nodes starting from the root of the DAG

The pending transaction pool contains all transactions that may affect a state variable in future blocks. HMS first scans this pool and returns only transactions related to the monitored state variables.

A transaction's parentage in the DAG is determined by its mark field. A transaction's *mark* is calculated in the following way:

$$mark_{current} \leftarrow h(mark_{previous}, val_{current})$$

¹<https://github.com/area67/sereth/tree/lock-free>

The function *h* is the Keccak256 hash function [11]. When a transaction *t1* is submitted, *mark_{previous}* is provided as a parameter. If *mark_{previous}* belongs to some transaction *t0*, then *t0* occurred before *t1*, and *t0* will be the parent of *t1* when inserted into the DAG.

Once a DAG is created from all pending transactions acting on the monitored state variables, the longest path of nodes is calculated starting from the root of the DAG. The transaction at the tail of this path is considered the latest state of the monitored variable. This heuristic is chosen to discourage participants from submitting transactions that would create branches in the DAG, as this reduces the overall number of potentially successful transactions per block.

III. METHODOLOGY

We implement lock-free HMS in Golang. An implementation within Geth is also provided with a simple use-case for measuring state-throughput.

The node struct is detailed by Algorithm 2. Each node corresponds to a blockchain transaction, and is created after parsing the transaction's input fields from the transaction pool. *mark* and *value* correspond to the parameters of the transaction when generated by a smart contract using HMS. *next* and *prev* store pointers to nodes that are the child or parent of the given node. *depth* is an integer storing the number of nodes between the root node of the graph and the given node.

The graph struct is given by Algorithm 1. *head* is a reference to node at the root of the tree, while *tail* references the node with the highest *depth*. *pending* is an array containing all nodes from the transaction pool which have been parsed and are currently being inserted into the list. When searching for the parent of a node, we check both the *pending* array and the DAG itself, which allows new nodes to be inserted in any order as long as a valid parent exists somewhere in either structure.

Algorithm 1 HMS Graph Structure

```

1: struct Graph
2:   Node head
3:   Node tail
4:   Node[] pending

```

Algorithm 2 HMS Transaction Structure

```

1: struct Node
2:   byte[] mark
3:   byte[] value
4:   Node[] next
5:   Node prev
6:   int depth

```

Before transactions can be inserted into the DAG, the transaction pool must be scanned for transactions with an HMS function signature. Transactions visible in the transaction pool

Algorithm 3 Method: Parse Transaction Pool

```
1: function PARSETRANSACTIONS(threadId, numThreads,  
   txns)  
2:   Node[] parsedList  
3:   interval  $\leftarrow$  (txns.size())/numThreads)  
4:   startIndex  $\leftarrow$  interval * threadId  
5:   for i  $\in$  [startIndex, startIndex+interval) do  
6:     if HMS_Signature  $\in$  txns[i].payload then  
7:       Node tx  $\leftarrow$  PARSE(txns[i].payload)  
8:       parsedList.append(tx)  
9:   Graph.pending.append(parsedList)
```

typically compress input parameters and function signature into a single byte string, or payload. This payload must be parsed in order to retrieve the smart contract function and input parameters.

Algorithm 3 details the PARSETRANSACTIONS method. We assume that *txns* contains all transactions that entered the system since the last call to PARSETRANSACTIONS. Unlike the previous implementation, Lock-free HMS does not need to parse the entire transaction pool on each call to PARSETRANSACTIONS, as any previously parsed transactions will be maintained in the DAG. The transaction pool is evenly divided among available threads. If a transaction with the correct function signature is found (Line 6), the input parameters for that method are parsed by extracting one or more hexadecimal substrings from the *payload* field (Line 7). On Line 9, we append each thread's results to a shared array. For this algorithm, we assume the APPEND operation is atomic and thread-safe. We require that PARSETRANSACTIONS execute completely for all given threads before attempting any INSERTNODE operations. This eliminates the possibility that a FINDPARENT operation on *txn₀* fails to locate *txn₀* since it has not yet been parsed.

Algorithm 4 Method: Find Parent

```
1: function FINDPARENT(Node txn)  
2:   for i  $\in$  pool.length do  
3:     curr  $\leftarrow$  Graph.pending[i]  
4:     if curr.mark == txn.prev_mark then  
5:       return curr  
6:   return FINDINGRAPH(txn, Graph.head)  
7: function FINDINGRAPH(Node txn, Node curr)  
8:   if curr.mark == txn.prev_mark then  
9:     return curr  
10:  for i  $\in$  curr.next do  
11:    item  $\leftarrow$  LOAD(curr.next[i])  
12:    return FINDINGRAPH(txn, item)  
13:  return null
```

Algorithm 4 gives the FINDPARENT method. In this method, we search the pool of nodes for the parent of the given transaction. If the transaction is not found in *pool*, we perform

a recursive search of the DAG by calling FINDINGRAPH. In this method, the *next* pointers of each node is atomically read, and then traversed.

Algorithm 5 Method: Insert Node

```
1: function INSERTNODE(Node txn)  
2:   parent = FINDPARENT(TXN)  
3:   if parent == null then  
4:     return false  
5:   txn.prev = parent  
6:   for i  $\in$  parent.next do  
7:     item  $\leftarrow$  LOAD(parent.next[i])  
8:     if item == null then  
9:       ret  $\leftarrow$  COMPAREANDSWAP(  
10:        &parent.children[i], item, txn)  
11:       if ret then  
12:         d  $\leftarrow$  LOAD(parent.depth)  
13:         if d != null then  
14:           FINISHINSERTING(txn, d + 1)  
15:       return true
```

Algorithm 6 Method: Finish Insert

```
1: function FINISHINSERTING(Node txn, int d)  
2:   depth  $\leftarrow$  LOAD(txn.depth)  
3:   if depth == null then  
4:     STORE(txn.depth, d)  
5:   for i  $\in$  txn.next do  
6:     item  $\leftarrow$  LOAD(txn.next[i])  
7:     FINISHINSERTING(item, d + 1)  
8:   if LENGTH(txn.next) == 0 then  
9:     while true do  
10:    Node tail  $\leftarrow$  LOAD(Graph.tail)  
11:    if txn.depth > tail.depth then  
12:      ret  $\leftarrow$  COMPAREANDSWAP(  
13:        &Graph.tail, tail, txn)  
14:      if ret == true then  
15:        break
```

Algorithm 5 gives the INSERTNODE method. First, the parent of the current node is located using FINDPARENT on Line 2. Afterwards, we iterate over the *next* array in an attempt to insert the new node as a child of the discovered parent node. If the COMPAREANDSWAP succeeds, the loop will terminate on Line 11. If the COMPAREANDSWAP fails, it means another thread has inserted a node at that index, so the iteration is continued. For this algorithm, we consider the *next* array to be unbounded.

On a successful insert, a node's depth must be calculated using algorithm 6. This algorithm checks the node's parent to see if it contains an initialized depth value and, if so, updates the node's own depth value. Afterwards we recursively call FINISHINSERTING to calculate the depth of all children of the node. If multiple threads attempt the STORE operation at Line

4, they are likely to overwrite one another. However, since the depth of a node is based on its parent, both thread will write the same value, thus this double write does not produce any anomalies. If FINISHINSERTING detects a leaf node whose depth is greater than the current *tail* (Line 11), it attempts to use COMPAREANDSWAP to update the tail of the graph. If a node’s parent does not contain an initialized depth value, it is possible that the current node is not yet reachable from the root of the graph. In this case, the work done by FINISHINSERTING will later be completed by a different operation which finds the node via the recursive call on Line 7.

IV. CORRECTNESS

In this section, we sketch a proof of the linearizability of our algorithm. An operation is linearizable if it appears to take effect instantaneously at some time t between its invocation and response [6]. This point in time t is often referred to as the linearization point of the operation. The linearization points of each method are as follows:

ParseTransactions. A call to PARSETRANSACTIONS with a set of transactions $txns_{new}$ and a transaction pool txn_{pool} such that $txns_{new} \in txn_{pool}$ linearizes the moment that the transaction pool is read prior to calling the method.

FindParent. Upon success, FINDPARENT with an input of txn_0 linearizes when a node txn_1 is loaded from either the *pending* array, or the *next* array of a node in the graph such that txn_0 is the child of txn_1 . Since FINDPARENT traverses each node in the *pending* array and each node reachable from the head of the DAG, if the method returns *null* on Line 12, there is some moment of time between the invocation and response of FINDPARENT at which txn_1 was not in the *pending* array or the DAG. An unsuccessful FINDPARENT linearizes at this point.

InsertNode. We differentiate “Node Reading” and “Tail Reading” operations. A *Node Reading* operation searches for a node in the DAG and returns *true* if it is found. A *Tail Reading* operation reads the current *tail* of the DAG. With respect to *Node Reading* operations, a successful call to INSERTNODE linearizes when the COMPAREANDSWAP operation on Line 9 succeeds. With respect to *Tail Reading* operations, a successful call to INSERTNODE linearizes when the FINISHINSERTING operation succeeds its COMPAREANDSWAP on the tail. Upon failure, INSERTNODE linearizes at the same moment as the unsuccessful call to FINDPARENT on Line 2.

We demonstrate the lock-freedom property of our algorithm by showing that for any number of threads n , at least one will make progress in a finite time. In PARSETRANSACTIONS and FINDPARENT, this is straightforward as each loop is bounded. In INSERTNODE and FINISHINSERTING, a thread may be suspended indefinitely by repeatedly failing the COMPAREANDSWAP operation. If this occurs, however, it means that another thread has succeeded their COMPAREANDSWAP operation, thus satisfying the condition that at least one thread has made progress in the system.

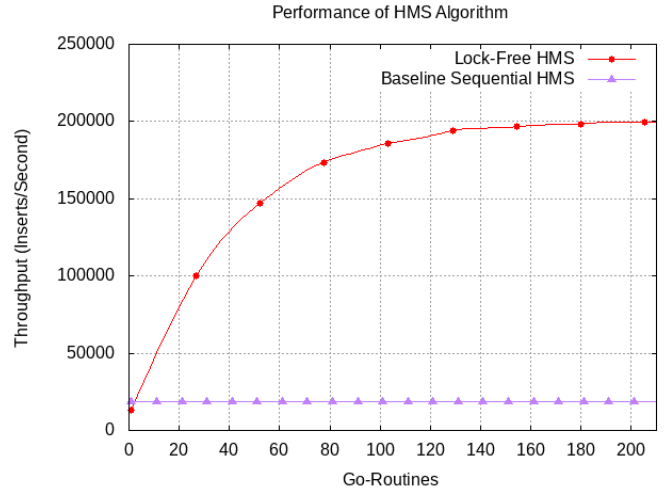


Fig. 1. Performance of HMS Algorithms

V. EXPERIMENTAL RESULTS

We test Lock-Free HMS against the sequential HMS algorithm on a 2GHZ AMD EPYC server with 32 cores and 64 threads. The test benchmark is executed on Ubuntu 18.04 LTS using go version 1.13. In each experiment we increase the number of go-routines (lightweight threads) until we no longer see a before increase. We measure the execution time to completely process a transaction pool containing 2^{15} transactions. We repeat each experiment 10 times and take the average execution time.

In order to eliminate the overhead of running full blockchain node, we create a test harness in Golang that simulates a transaction pool. The test harness supplies the HMS algorithm with a transaction pool struct identical to the one used in Go Ethereum. Execution begins after the transaction pool has been generated, and concludes after all transaction have been inserted into the DAG, and the longest chain has been calculated.

In Figure 1, we plot the performance of the sequential algorithm on a single thread as a baseline for comparison. Lock-free HMS gains performance steadily as the number of go-routines increases. Up to 128 go-routines, we see an average increase in throughput of 6.4x. Beyond 128 go-routines we see very little performance gain on our system. At roughly 200 go-routines, we see a maximum throughput increase of 11.1x. Due to some overhead in the Lock-Free algorithm, the sequential implementation outperforms Lock-Free HMS at 1 and 2 go-routines.

VI. CONCLUSION

In this paper we presented an efficient lock-free implementation of the Hash-Mark-Set algorithm. Our algorithm achieves significant speedup over the existing implementation and improves the speed at which an HMS contract is able to provide a response regarding the value of a state variable.

REFERENCES

- [1] Iddo Bentov, Charles Lee, Alex Mizrahi, and Meni Rosenfeld. Proof of activity: Extending bitcoin's proof of work via proof of stake. *IACR Cryptology ePrint Archive*, 2014:452, 2014.
- [2] Miguel Castro, Barbara Liskov, and Others. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- [3] Victor Cook, Zachary Painter, Christina Peterson, and Damian Dechev. Read-Uncommitted transactions for smart contract performance, 2019.
- [4] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkeley, CA, 2017.
- [5] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.
- [6] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [7] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper*, August, 19, 2012.
- [8] Douglas MacGregor, David S Mothersole, and John Zolnowsky. Method and apparatus for a compare and swap instruction, April 1986.
- [9] Jim Melton and Alan R Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.
- [10] Satoshi Nakamoto and Others. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [11] 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.
- [12] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), September 1997.
- [13] Gavin Wood and Others. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.