IEEE ICBC'24 1570981734

Lock-Free Concurrent Smart Contracts

Authors Anonymized

***Abstract*—Smart contracts are commonly used on blockchains to define complex behaviors not possible with simple transactions. Miners and validators could achieve significant performance gains by executing smart contracts in parallel, but validators must be able to re-execute the proposed block deterministically. Existing works capture the execution order of concurrently executed transactions using locks, or Software Transactional Memory, enabling validators to re-execute the block deterministically. These approaches can introduce a costly overhead for large blocks, in which many transactions must be compared to deduce their ordering. In this paper, we present a methodology for executing smart contracts concurrently without locks by associating each smart contract state variable with a descriptor, and updating these descriptors with Compare-And-Swap (CAS). Whenever a descriptor is updated, we create a reference to the previous descriptor within the new one. The resulting graph represents all conflicts between transactions, and can be used to deterministically re-execute the transactions in parallel. Our approach captures the ordering between transactions whenever a semantic conflict is detected, eliminating the need to compare lock acquisitions, or timestamps. Additionally, our approach provides a guarantee of Lock-Free progress. In our experimental evaluation, our approach achieves a maximum speedup of 70x against related work when tested on block sizes similar to that of Bitcoin, or Ethereum.**

I. Introduction

A blockchain is a distributed network where participants choose blocks of transactions to be appended to a public ledger. These blocks are secured via cryptographic link to minimize the chance that blocks can be modified in the future. The core function of blockchain platforms, such as Bitcoin [17] or Ethereum [19], is to allow participants to exchange digital currency. Users do this by submitting a transaction, which identifies the address of the recipient, as well the quantity of currency to send. The functionality of a transaction is limited to transfers of cryptocurrency from one party to another. To address this, modern blockchains implement smart contracts. A smart contract is a set of functions and state variables which are appended to the ledger. Once in the ledger, smart contract function calls are executed by miners, who write the request and corresponding change in state back to the ledger. Some smart contract languages such as solidity [18] are Turing-complete, enabling advanced behavior not possible with regular blockchain transactions. Smart contracts can declare "state variables", the values of which are written to the ledger. Smart contracts can write changes to the state variables, and even make calls to methods in other deployed smart contracts.

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 smart contracts concurrently, both 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 interleavings 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. [6] propose an approach based on "Transactional Boosting" [10], 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 approach, all locking operations are logged in order to generate a deterministic schedule for the concurrently executed transactions. Deducing this schedule requires comparing the set of locks acquired by each transaction, requiring $O(n^2)$ time, where n is the number of transactions per block. This results in a reduction in performance as block size increases. Furthermore, in the case of conflicts, transactions must be aborted and physically rolled back. This requires miners to spend time re-executing transactions without additional compensation from the smart contract fees. Finally, locks cannot guarantee system-wide progress [11].

Anjana et al. [2] propose "OptSmart," an algorithm which utilizes timestamp-based Software Transactional Memory (STM). The concurrent schedule is generated using the timestamp ordering given by the underlying STM. This approach

has similar drawbacks, in that it must compare the timestamps of transactions operating on the same object x in order to deduce their ordering, and generates spurious aborts when transactions conflict.

Existing blockchains, such as Bitcoin, support blocks sizes of 1MB. Some blockchains, such as BSV, support blocks up to 4GB in size, close to 4,000 times larger than that of Bitcoin. For this reason, we aim to implement an approach that captures the runtime ordering between transactions based on the semantic conflicts, without having to compare lock ordering or timestamps, in order to achieve greater scalability in networks with large blocks of transactions. Furthermore, an ideal approach reduces the number of spurious aborts that occur during transaction conflicts, in order to reduce the number of instances where miners execute transactions without compensation from smart contract fees.

Lock-freedom is a progress guarantee that ensures that given multiple threads executing concurrently, at least one thread will make progress in a finite amount of time [11], [8]. Lock-Free algorithms are difficult to design, but have the potential to create large speedups over lock-based algorithms. Lock-Free algorithms often utilize Compare-And-Swap (CAS), and descriptor objects to facilitate thread synchronization without the use of locks. CAS is an atomic instruction that, given a memory address and a value, will update the memory address only if the given value matches the current value of the memory address. If multiple threads attempt to update a memory address concurrently using CAS, only one will succeed. A descriptor object [5] is an object that contains all information necessary for an arbitrary thread to complete an operation. Descriptor objects are often utilized in situations where more than one memory location needs to be updated in a single atomic step without incurring the expense of a multi-word CAS [7], [16].

In this paper, 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" [20]. We protect each smart contract state variable with a reference to a descriptor, which is updated using Compare-And-Swap. In order for a smart contract transaction to commit, a thread must update the descriptor reference for each state variable associated with the transaction. Threads help with the execution of conflicting transactions by reading their descriptor objects. This approach prevents transactions from aborting spuriously, and reduces the number of transactions which must be re-executed. Additionally, we capture the ordering of concurrently executed transactions by including a reference to the previous descriptor' transaction within each newly generated descriptor. The resulting graph represents a strictly serializable history of all executed transactions, and is constructed without the use of any additional logging operations. An overview of this approach is given by figure 1. Smart contract code is modified to contain a descriptor pointer for all state variables. Each

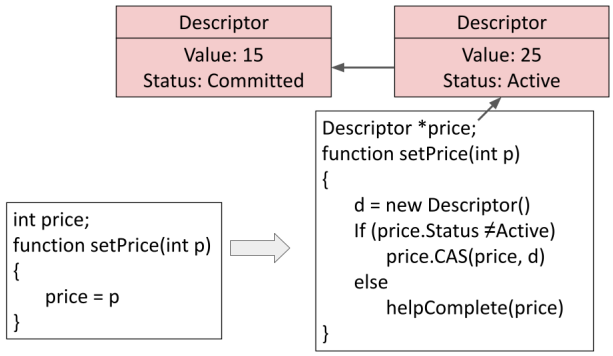

Fig. 1. Descriptors protect state variables, enabling thread synchronization and naturally capturing transaction conflicts

descriptor contains information about the operation occurring at its corresponding state variable, in addition to a reference to the smart contract that the operation belongs to. This allows threads to identify when a conflicting transaction is taking place at a shared state variable. In the case that a thread finds a descriptor that is part of an active transaction, the thread uses the descriptor object to help the conflict transaction complete. Afterward, the thread uses CAS to update the descriptor pointer. 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 [6]. Afterward, validators execute the same algorithm, obeying any total orderings given by G. If the validator produces a result that differs from G, the proposed block is invalid.

Our approach addresses the problem of spurious aborts during transaction conflicts with the use of its thread helping scheme. In our approach, transactions only ever abort if they encounter a cyclic dependency, which we demonstrate to be far rarer than in related approaches. Furthermore, our approach generates a graph of all conflicts by repurposing the descriptors created during transaction execution. This avoids the use of locks or STM to capture the parallel ordering of transactions, and does not require any comparison of lock acquisitions, or timestamps. These contributions perform well in cases where smart contract calls are not unlikely to conflict, and in cases where block contain many transactions, such as in Bitcoin, or BSV.

We implement our approach in C++ and compare against the work of Dickerson et al.

The contributions of this work are as follows:

1) We present an efficient lock-free approach for concurrent execution of smart contracts, in which threads can co-operate to resolve transaction conflicts without aborting.
2) We present a novel use of descriptors as a method for efficiently capturing a concurrent execution without

reliance on locks or STM.

3) We implement our approach in C++, and provide direct comparisons against the work of [6]. Due to the efficiency of our descriptor-based graph, our approach achieves a speedup as high as 70x for block sizes similar to that of Bitcoin, or Ethereum.

II. Related Work

Several works address the sequential execution of blockchain smart contracts. Dickerson et al. [6] propose an algorithm that enables miners to execute transactions concurrently using mutual exclusion locks. The authors define *storage operations* as primitive operations on a state variable. Each storage operation is protected by an abstract lock. In order to execute a smart contract, a thread executes the smart contract code, stopping before each storage operation to acquire the associated lock. If the lock is held by another thread, it means there is another smart contract execution is currently accessing that state variable, and the thread must wait. If the thread acquires the lock, it executes its storage operation and proceeds to the next. Once all storage operations are complete, the thread releases all locks that it is holding. In order to ensure that validators can reproduce the results of this concurrent execution, each time a thread acquires a lock, it also increments a counter in the lock. The value of each counter during each transaction's execution can be used to determine in which order conflicting transactions executed. This is used to create a fork-join schedule, which is distributed to validators. Despite pessimistically acquiring locks, the approach must resolve transaction conflicts by aborting one of the conflicting transactions. In our approach, threads are synchronized using descriptors instead of locks, enabling threads to work cooperatively on any arbitrary transaction. Furthermore, we utilize the CAS-based update of descriptor objects to capture the execution order of conflicting threads, rather than performing a lengthy comparison of lock counters.

Anjana et al. [1], [2] propose an approach based on software transaction memory (STM). This approach attempts to execute each piece of smart contract code using a "Multi-Version Timestamp Ordering" (MVTO) STM [14], and uses the timestamps to generate a graph of transaction conflicts. MVTO compares the timestamps of conflicting transactions, and aborts one if the timestamps overlap. This approach uses multi-versioning to prevent read-only transactions from aborting during conflicts. Jin et al. [13] present a similar approach based on Optimistic Concurrency Control (OCC). These approaches both suffer from extraneous aborts in high contention workloads, creating a bottleneck for concurrent performance. Our approach differs in that our approach is not optimistic. Descriptors are acquired one-by-one as each state variable is accessed. This enables threads to resolve conflicts by helping the conflicting transaction complete instead of aborting.

Gao et al. [9] propose an approach that divides users on the network into groups, and distributes a subset of smart contracts to each group. In order to prevent users from gaining disproportionate representation amongst the groups by generating a large number of pseudonyms, only users that have published a block within the last l blocks are eligible. Additionally, a two-level load balancing scheme is proposed to ensure that each group is incentivized to work on their own set of assigned smart contracts. This approach addresses the problem by partitioning the workload, and executing each partition sequentially. Our approach differs in that all transactions are executed concurrently by all validators.

There are several works dedicated to the problem of deterministically executing a parallel history [4], [1], [6]. Our algorithm is designed to generate a strictly serializable schedule to guarantee compatibility with any approach.

III. 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.

A. Primary Execution

Algorithm 1 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
```

Algorithm 1 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 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 [15] 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 2 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 2 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.

Algorithm 3 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)
```

Transaction execution is handled with an approach based on Lock-Free Transactional Transformations [20], and is given by Algorithm 3. This algorithm takes a descriptor, and executes the 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 3.8, 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 3.15, 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 4. This method takes a state variable pointer *p*, and performs the operation given by *type*, utilizing any arguments given by *args*. On line 4.2, we check if the current transaction is still active by check the status field of its descriptor. Since 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 4.4. 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

Algorithm 4 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
```

Algorithm 5 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
```

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 4.17. Additionally, the helpstack is updated to increment the opid of the current transaction.

Algorithm 5 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 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 *ExecutTransaction* on line 5.4. Otherwise, if another thread has already completed the operation, the rest of the operation is skipped (line 5.6). The current value of the state variable is logically interpreted based on the status of the descriptor on line 5.7. 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 5.21. If the CAS succeeds, it means no other threads attempted to update p after its value was read on line 5.2. 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 5.20.

B. 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 6 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 6 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 6.7. 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.

The *ValidateBlock* method takes a graph of descriptors given

Algorithm 7 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)
```

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 7.4. In the *Validate* method, the block is re-executed deterministically using a fork-join approach. On line 7.10, 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 7.12 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 7.6. 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.

IV. CORRECTNESS

Our proposed algorithm is designed for the correctness condition strict serializability. The definition of strict serializability is given by Herlihy and Koskien [10]. Our work follow from the proofs of Zhang et al. [20] 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 [12], 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*

The linearization points for a variable updated in the style of algorithm 2 occurs on line 5.21, 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*

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 5.21. 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 5.4, 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 5.21 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 [10], given Lemma 1 and 2, our proposed algorithm is strictly serializable.

A. 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 4. 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 5.21 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 5.21, and at least one will make progress.

B. 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 5.2. If the CAS operation on line 5.21 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 5.20. 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.

V. 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 two experiments in order to analyze scalability of concurrent smart contract execution, as well as the performance of the graph construction algorithm for both approaches.

In the first experiment, 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, providing a view of how each approach scales. 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 our second experiment, we measure the performance of each algorithm, including the overhead of graph building for real-world block sizes. We vary the size of a block of transactions from 400 to 2,000 to align with that of Bitcoin.

In order to compare our approach, we implement the transactional boosting approach of Dickerson et al. [6]. 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 2 gives the results of our throughput experiments. We denote our approach "LFTT," and transaction boosting approach "Boosting." We observe that without including the overhead of graph construction, both algorithms perform similarly. 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 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.

Figure 4 gives the results of our experiments for real-world block sizes, including the cost of graph building for both algorithms. In this experiment, the number of threads remains constant at 8, and instead, the number of transactions in the block is placed on the x-axis. In this experiment, the difference between the $O(n^2)$ graph construction algorithm of Boosting is evident. In addition to logging each locking operation during execution, the lock profiles of each transaction must be compared to determine if they acquired the same locks, and if so, which transaction observed a smaller value in the lock counter. In LFTT, transaction conflicts are detected directly when CAS is used to change the *OpInfo* at a memory location. When this occurs, a pointer can be included in the new descriptor in $O(1)$ time. Constructing the graph this way requires only $O(n*m)$ time, where n is the size of the block, and m is the number of *OpInfo* created per transaction. As the size of the block grows, Boosting degrades in overall performance, while LFTT actually achieves some speedup. This is due to the larger block size giving threads a longer time to spawn, and work concurrently. For a block size of 2,000

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

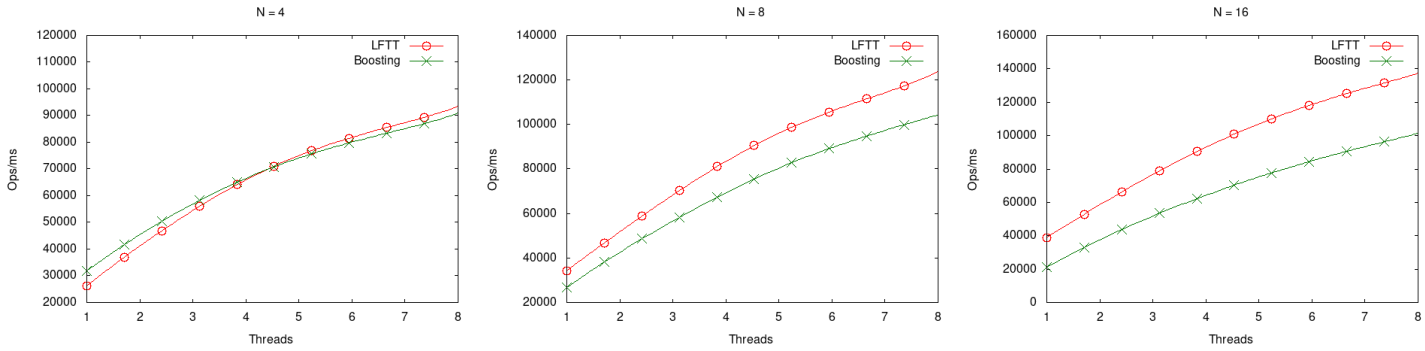

Fig. 2. Scalability Comparison

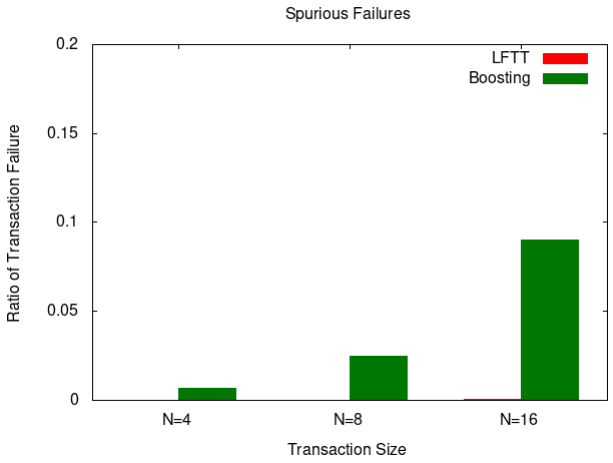

Fig. 3. Failed Transactions Comparison

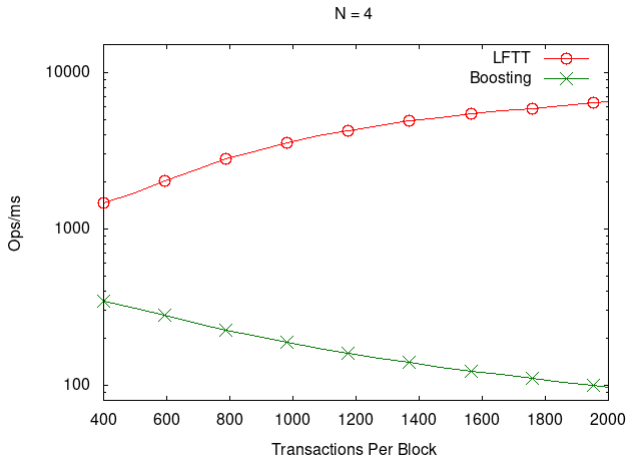

Fig. 4. Graph-Building Comparison

transactions, LFTT achieves a speedup of 70x in comparison to boosting.

VI. Conclusion

In this paper, we present 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. Our approach achieves a speedup of 70x over related works for block sizes similar to those of Bitcoin, or Ethereum.

References

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

[2] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. Optsmart: a space efficient opt imistic concurrent execution of smart contracts. *Distributed and Parallel Databases*, pages 1–53, 2022.

[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] Robert L Bocchino, Vikram Adve, Sarita Adve, and Marc Snir. Parallel programming must be deterministic by default. *Usenix HotPar*, 6(10.5555):1855591–1855595, 2009.

[5] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 190–199, 2001.

[6] 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.

[7] Steven Feldman, Pierre LaBorde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, 43:572–596, 2015.

[8] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[9] Zhimin Gao, Lei Xu, Lin Chen, Nolan Shah, Yang Lu, and Weidong Shi. Scalable blockchain based smart contract execution. In *2017 IEEE 23Rd international conference on parallel and distributed systems (ICPADS)*, pages 352–359. IEEE, 2017.

[10] 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, PPoPP '08, pages 207–216, New York, NY, USA, February 2008. Association for Computing Machinery.

[11] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

[12] Maurice P 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.

[13] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering*, 34(11):5070–5083, 2021.

[14] Priyanka Kumar, Sathya Peri, and K Vidyasankar. A timestamp based multi-version stm algorithm. In *Distributed Computing and Networking: 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings 15*, pages 212–226. Springer, 2014.

[15] 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.

[16] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 314–323, 2003.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[18] Solidity. https://soliditylang.org/. Accessed: 2023-12-22.

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

[20] 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.