SC-CHEF: Turboboosting Smart Contract Concurrent Execution for High Contention Workloads via Chopping Transactions

Dibo Xian and Xuetao Wei, *Member, IEEE*

***Abstract*—Concurrent execution of smart contracts is a promising approach to boost their performance in current blockchain systems. However, prior work on concurrent execution only considered each smart contract transaction as a single atomic unit for the concurrent scheduling, which is not suitable for the increasingly complex smart contracts with high contention in practice. In this article, we present SC-CHEF, a new concurrency control system to turboboost smart contract concurrent execution for high contention workloads via the transaction decomposition. The key novelty of our SC-CHEF is to decompose the original transaction into multiple subtransactions that can be executed independently, while still respecting the original logic dependency. We implement and evaluate our SC-CHEF in the private Ethereum platform. Our extensive experiments show that our SC-CHEF outperforms the classic optimistic concurrency control and two-phase lock by 60% and 70% on average in the case of high contention, respectively.**

***Index Terms*—Concurrency, high contention, smart contracts, transaction decomposition.**

I. INTRODUCTION

AS A major advancement in blockchain technology after Bitcoin, Ethereum [1] combines smart contracts [2] and blockchain technology to give the blockchain programmable features, enabling blockchain technology to be applied to a wide range of applications other than the field of cryptocurrencies. The smart contract is a computer program stored in the blockchain that describes contract terms and agreements executed automatically, and its integrity is guaranteed by the blockchain technology.

With the evolution of blockchain technology, blockchain systems can be divided into two categories: 1) permissionless blockchain and 2) permissioned blockchain [3], which can be applied to different scenarios. The main bottleneck of these two kinds of blockchain is performance issues. On the one hand, the high cost consensus mechanism of blockchain, such as proof of work (PoW) [4] and Practical Byzantine Fault Tolerance (PBFT) limit its scalability . On the other hand, the serial execution mode of smart contracts limits the complexity of the business that the blockchain can support. Some work uses sharding technology to scale the performance of the blockchain at the consensus layer. However, the concurrency capabilities of the smart contract execution layer have yet to be explored.

Furthermore, with the expansion of application fields of the blockchain, the business logic described by smart contracts is becoming more and more complex, which introduces the significant time cost of the execution of smart contracts. On the other hand, each smart contract possesses an independent storage space called contract account, where the code and state variables of the smart contract are stored. A mature smart contract application project with good maintainability must carefully organize the relationship between smart contract logic and data. For example, separately deploying and storing business logic code and data in different smart contract accounts can simplify the cost of shared data maintenance [5], [6]. Thus, the behavior of smart contracts frequently calling other contracts is becoming very common in the blockchain's ecosystem [7]. In order to improve the performance of the blockchain, it is critical to explore how to accelerate the execution of the increasingly complex smart contracts with high contention.

There exists some study on accelerating the execution speed of smart contracts by adding concurrency to smart contracts execution [8], [9], [10], [11]. However, these studies were mainly designed on the basis of variants of optimistic concurrency control (OCC) [12] and two-phase lock (2PL) [13], which are essentially severely affected by varied contention workloads in practice. Furthermore, they only considered each smart contract transaction as a single atomic unit for the concurrent scheduling.

In order to fully explore the performance of concurrent execution of smart contracts for high contention workloads in the blockchain while achieving concurrent scheduling with serializability and recoverability (see Section II-C), we propose SC-CHEF, a new concurrency control system that boosts concurrent execution of smart contracts by decomposing transactions. In SC-CHEF, each transaction will be first decomposed into multiple Tx-pieces that can be executed independently according to the calling relationship of its corresponding smart contract, while still respecting the original logic dependency (Note that, each Tx-piece corresponds to a call to a smart contract.). Then,

Manuscript received 15 July 2022; revised 20 December 2022 and 26 March 2023; accepted 7 July 2023. This work was supported in part by the National Key R&D Program of China under Grant 2021YFF0900300, in part by the Key Talent Programs of Guangdong Province under Grant 2021QN02X166, and in part by the National Natural Science Foundation of China (Project No. 72031003). Associate Editor: J. Zhou. *(Corresponding author: Xuetao Wei.)*

The authors are with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: 11930389@mail.sustech.edu.cn; weixt@sustech.edu.cn).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2023.3296278.

Digital Object Identifier 10.1109/TR.2023.3296278

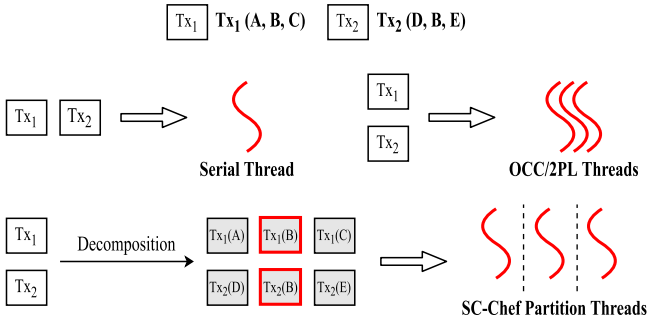


Fig. 1. Comparison between Serial (Serial Execution), OCC (Optimistic Concurrency Control), 2PL (Two Phase Lock) and our SC-CHEF transaction execution processes. Tx_1 and Tx_2 are conflicting transactions. SC-CHEF enables them to be executed concurrently through transaction decomposition because only two of the six subtransactions are in conflict.

these Tx-pieces will be allocated to corresponding concurrent threads according to their corresponding smart contract account addresses. Each thread will organize its assigned Tx-pieces into a dependency graph in the form of a directed acyclic graph, and each node of the graph will be traversed and executed when the serializable condition is met.

An illustration example is also given in Fig. 1. There are two transactions Tx_1 and Tx_2. Tx_1 accesses smart contracts SC_A, SC_B, SC_C in sequence, and Tx_2 accesses smart contracts SC_D, SC_B, and SC_E in sequence. Tx_1 and Tx_2 are in conflict because they all access smart contract SC_B. In the serial execution mode, these two transactions can only be executed serially. These two conflicting transactions cannot be executed concurrently by the OCC/2PL threads, where one of them must be aborted and reexecuted or blocked until the other finishes execution. However, in SC-CHEF, transactions are first decomposed into multiple Tx-pieces according to the smart contracts they call. In this example, Tx_1 and Tx_2 are decomposed into six Tx-pieces, and only two Tx-pieces that access smart contract B are in conflict. Then the SC-CHEF threads will execute these Tx-pieces. Since only two of the six Tx-pieces are in conflict, the remaining Tx-pieces can be executed concurrently by multiple threads. SC-CHEF achieves fine-grained concurrency control through transaction decomposition.

We implemented and evaluate our SC-CHEF system in the private Ethereum platform [14]. For comparison, we also implement the classic OCC and 2PL concurrency control system in the private Ethereum platform. In our experiments, we compare the performance of our SC-CHEF with that of OCC, 2PL, and the original serial method of Ethereum. Extensive experiment results demonstrate that our SC-CHEF is 4.6 times faster than the serial execution method. Furthermore, SC-CHEF outperforms the classic OCC and 2PL by 60% and 70% on average, respectively, under high contention workloads, under the premise of guaranteeing serializable and recoverable scheduling. All these promising results shed the light on exploring benefits of smart contract concurrent execution via the transaction decomposition in the blockchain, in order to meet practical performance requirements from real-world applications.

In a nutshell, our contribution can be summarized as follows.

1) We propose SC-CHEF, a new concurrency control system based on the transaction decomposition, which significantly boosts concurrent execution of increasingly complex smart contracts in the blockchain.
2) We propose to decompose the original transaction into multiple subtransactions that can be executed independently, while still respecting the original logic dependency.
3) We prototype our SC-CHEF system and implement it in the private Ethereum platform. Our extensive experiments demonstrate that our SC-CHEF outperforms the classic OCC and 2PL by 60% and 70% on average, respectively, under high contention workloads.

The rest of this article is organized as follows: In Section II, we introduce the background and motivation. Then, we present our system SC-CHEF in detail in Section III. We evaluate SC-CHEF in Section V and discuss the related work in Section VII. Finally, Section VIII concludes this article.

II. BACKGROUND AND MOTIVATION

In this section, we first introduce the background of blockchain and smart contracts. Then, we draw our motivation based on the concurrency issues of smart contracts.

A. Blockchain

Blockchain technology is originated from the Bitcoin digital cryptocurrency proposed by Satoshi Nakamoto in 2008 [15]. Bitcoin is a distributed ledger that records cryptocurrency transactions. In order to expand the application areas of blockchain, the Ethereum [1] achieved a great advancement in blockchain technology by enabling blockchain to be programmable with smart contracts. The global replicated consensus architecture of the blockchain requires each node to store the same smart contract data. Each transaction needs to be executed repeatedly by each node. The global replicated consensus architecture of the blockchain requires each node to store the same smart contract data and each smart contract transaction to be executed repeatedly by each node, which results in scalability issues.

B. Smart Contract

In 2015, the Ethereum [1] achieved a great advancement in blockchain technology by combining blockchain technology with smart contracts. The term smart contract was proposed by Nick Szabo in 1990s [2], which is a computer program that describes automatically executed and controlled contract terms and agreements. At that time, smart contracts were not legally binding and unable to create a credible trading environment between users without a third-party authority. The emergence of blockchain technology enables smart contracts to be executed in a secure environment. At the same time, smart contracts make blockchains programmable, thus expanding application areas of blockchains.

Smart contracts can take many forms and execution models on the blockchain. Here, we explain the execution process of smart contracts based on Ethereum. Ethereum uses the account-based data model, which includes external owned accounts and contract accounts. External owned accounts represent users of

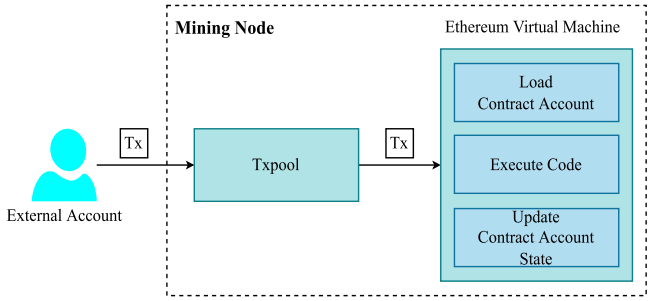


Fig. 2. Execution process of a smart contract transaction in Ethereum. When a transaction that calls a smart contract is executed by a mining node, the code in the corresponding smart contract account will be executed in the EVM, after which the state of the account will be updated.

the blockchain. Each contract account stores the code and data of a smart contract deployed by the external owned account. Normally each smart contract provides a set of functions, also called interfaces, for external owned accounts to read or modify the data in the contract accounts. A transaction in Ethereum represents a contract creation, a transfer action, or a call to a smart contract, which may change the state variables of the contract account. Fig. 2 shows the execution process of a transaction that calls a smart contract. The external owned account constructs a transaction containing the address of the contract account and the necessary input parameters of the smart contract to be executed. Transactions initiated within a period of time will be broadcast to mining nodes and stored in their transaction pools. In the block generation cycle of the mining nodes, transactions will be taken out and executed in the Ethereum virtual machine (EVM) one by one.

C. Smart Contract Concurrency Issue

Originally, smart contracts developed in the blockchain are quite simple. Compared with consensus overhead, the benefits of concurrent execution of these simple smart contracts are not significant. Thus, these blockchains simply choose to execute transactions one after another in a serial mode.

With the evolution of the blockchain ecosystem, business-oriented smart contracts are becoming the main workload. Furthermore, with the expansion of blockchain application fields, the business logic described by smart contracts is becoming more and more complex, which results that the cost of executing smart contract transactions is nontrivial. The situation becomes even worse when smart contract transactions are executed serially currently. The major problem of the mechanism of serially executing transactions is that it greatly limits the throughput of the blockchain. Therefore, it is critical to add the concurrency for the smart contract transaction execution to improve the performance of the blockchain systems.

The concurrent execution of multiple smart contract transactions will cause data contention, which may lead to inconsistent final state of blockchain if without transactions' concurrent schedule. For example, there are two different transactions trying to call the same smart contract and modify its state. Suppose these two transactions are executed at the same time, the final state of the smart contract depends on the order in which these two transactions successfully modify the state of the contract. Concurrency control systems, such as OCC [12] and 2PL [16] have been proposed to obtain deterministic transaction concurrency scheduling in traditional database systems. These systems are mainly designed based on two principles: 1) serializability and 2) recoverability.

Serializability: Serializability is an isolation level for concurrent execution of transactions that meets the requirement of data consistency. If multiple transactions access or modify common data objects at the same time, the execution order between these conflict transactions must be determined. For these concurrent transactions that access common data objects, if the execution result is equivalent to a certain serial execution result, the schedule is serializable.

Recoverability: The final state of a transaction execution is either commit or abort. If a transaction commits, its modifications to data objects will be persisted in the database. Conversely, if a transaction aborts, its modifications to data objects will not take effect, which means that all of the transaction's writes will be rolled back. The most important thing is that a recoverable scheduling system must ensure that these aborted modifications will not be read by other committed transactions.

In short, serializability restricts the execution order of multiple conflicting transactions, and recoverability restricts the visibility of write operations between concurrent transactions. All concurrency control systems that can be applied to blockchain systems must meet these two characteristics. To achieve serializability and recoverability, both OCC and 2PL adopt different lock mechanisms to strictly limit concurrent transactions' access to the shared memory. In 2PL, each transaction applies for exclusive locks for the written objects and holds them until the end of the transaction. During this period, other transactions cannot read or write the locked objects. For OCC, transactions write to their temporary local caches. These writes will be persisted to the actual shared data objects only after the validation .

Inspired by the concurrency control of traditional database systems, in order to improve the performance of blockchain, there was some work on adding concurrency to the execution of smart contracts. Dickerson et al. [8] proposed a 2PL-based method which allows mining nodes to execute smart contracts concurrently. Smart contracts are executed on multiple threads, while the serializable schedule is achieved by managing the allocation of abstract locks, which are related to storage operations. Anjana et al. [9] applied the idea of OCC to the smart contract execution, which improves the performance under low contention workloads. Pang et al. [10] proposed a smart contract concurrent execution solution combining OCC and batch process. During the mining phase, transactions that decrease the overall performance are aborted to optimize the performance of mining and verification nodes. All of these algorithms above rely on the directed acyclic graph generated by the mining node to enable the verification node to deterministically replay the transactions in the block so as to reach a consistent final state.

Each smart contract possesses independent storage space, while one transaction can operate multiple contracts. High data contention means that a small number of hot smart contracts are

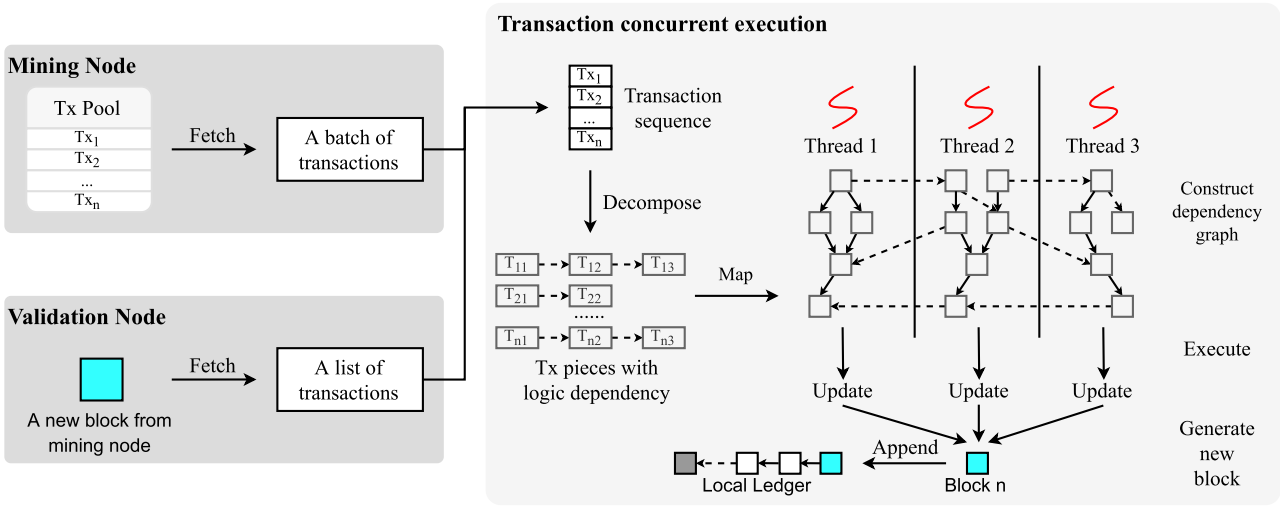


Fig. 3. Overview of SC-CHEF.

operated by a large number of transactions, and these conflicting operations must be serialized to ensure serializability.

Previous work was mainly designed on the basis of variants of OCC and 2PL, which are not only essentially severely affected by varied contention workloads, but also simply regarded each smart contract transaction as a single atomic unit for the concurrent scheduling [17]. However, a smart contract transaction is becoming more and more complex as its business logic involves multiple smart contracts to interact with each other to finish the contract's objectives [5]. Namely, during the execution of a smart contract, the code and states of other smart contracts are often called and accessed by this running smart contract [18], [19]. It is inefficient for the concurrent scheduling to only consider each smart contract transaction as a single atomic unit. Therefore, in this article, we investigate a concurrency control system to coordinate the concurrent execution of conflicting smart contract transactions to achieve both serializable and recoverable schedule by decomposing transactions.

III. OUR APPROACH: SC-CHEF

In this section, an overview of SC-CHEF is first provided. And then we describe our SC-CHEF in detail.

A. Overview

Traditional concurrency control protocols cannot efficiently handle varying contention workloads due to random aborts or replays caused by data contention. Inspired by deterministic concurrency control protocols [20], [21], combined with the execution characteristics of the smart contract transaction, our SC-CHEF generates deterministic concurrent scheduling of smart contract transactions.

Fig. 3 provides an overview of SC-CHEF, which runs in all mining and validation nodes in the blockchain system. The process of concurrent execution of transactions in SC-CHEF is as follows.

1) During the transaction execution phase, each node takes a batch of ordered transactions as input to SC-CHEF.
2) Transactions are decomposed into multiple subtransactions based on the read and write sets declared in advance.
3) These subtransactions are mapped to the corresponding partition threads based on the data they access.
4) For each partition thread, whenever it is assigned a new subtransaction, this thread updates its local dependency graph based on the read and write relationships between the subtransactions.
5) After all partition threads have completed the construction of their respective dependency graphs, each partition thread starts executing the subtransactions in the order of the topology order of its local dependency graph.
6) Finally, a new block is generated by integrating the states of all the partitions.

B. Transaction Decomposition

Unlike conventional concurrency control systems that do not consider the logic within transactions, SC-CHEF explores the concurrency ability of smart contract execution logic. Namely, SC-CHEF decomposes the original transaction into multiple subtransactions that can be executed independently while still respecting the original logic dependency.

Some techniques of static analysis, such as control flow graph [22], can be applied to transaction decomposition, which can decompose the transaction's code into a directed acyclic graph composed of statements according to its logic. However, it is difficult and inefficient to decompose transactions of smart contracts using the above method for the following reason. Smart contracts can be written in many Turing complete programming languages, such as Solidity or Serpent [23] in Ethereum. No matter which language is used, smart contracts will eventually be compiled into bytecode that can be executed in the EVM. Since EVM is a virtual machine based on the stack structure, the execution of bytecode similar to assembly language is highly dependent on its current context. Therefore, decomposing transactions at smart contract's statement level will not enable subtransactions to be executed independently.

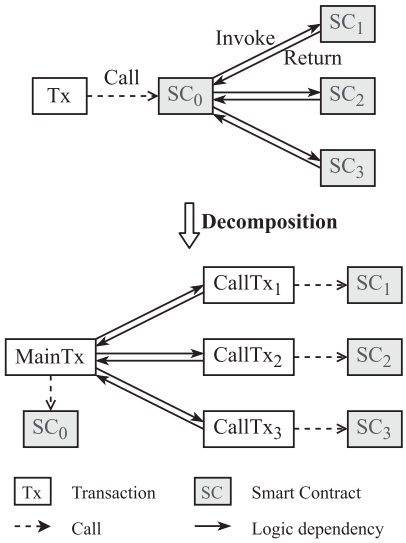


Fig. 4. Decomposition process of a transaction based on the calling logic between contracts. A transaction represents a call to a smart contract, and there are also internal calls between smart contracts. SC-CHEF decomposes a single transaction into multiple TX-pieces according to the internal call relationship between smart contracts.

In the blockchain system, each smart contract contains independent storage space and code. The interaction between smart contracts is realized by calling each other's interfaces, like functions being called in a sequential program. In the previous introduction, a mature smart contract application project with good maintainability must carefully organize the relationship between smart contract logic and data [5]. For example, separately deploying and storing business logic code and data in different smart contract accounts can simplify the cost of shared data maintenance. Therefore, the behavior of smart contracts frequently calling other contracts is becoming very common in the permissioned blockchain's smart contract ecosystem.

In the proposed system, all smart contracts are allocated to multiple mutually exclusive partitions according to contract addresses, where each partition corresponds to a thread that is responsible for the execution of smart contracts within the partition. Invocations between smart contracts within a transaction can be mapped to interactions between threads. Our transaction decomposition method is based on call relationships between smart contracts, which decomposes each transaction into multiple subtransactions that can be independently executed in the EVM. Fig. 4 shows the process of a transaction being decomposed. A transaction (Tx) calls smart contract 0 (SC_0). During the execution of SC_0, it will call SC_1, SC_2, and SC_3 in turn. SC_0 can be regarded as the main logic of an application, the other three contracts store different types of data, and the calling behaviors can be regarded as the operations of reading and writing data. During each call, SC_0 will wait until the called contract finishes its own execution and returns the execution result. The key idea of transaction decomposition is to map the invocation relationship between contracts to the logical dependency between subtransactions. As shown in Fig. 4, the original transaction Tx is decomposed into a MainTx and multiple CallTxs. The MainTx is corresponding to the smart contract that is directly called by the original transaction. The CallTx is corresponding to the smart contract that is called by the MainTx's corresponding smart contract. The logical dependency between MainTx and CallTxs is the same as the call relationship within the original smart contract. The execution process of the decomposed transaction is shown in Fig. 5. At the beginning, MainTx calls SC_0 and executes its code. When it reaches the statement that calls SC_1, the thread to which the SC_0 resides will suspend the MaxinTx and send an invocation message with call parameters to the thread to which the SC_1 (executed by CallTx_1) resides. Only when the CallTx_1 of the SC_1 finishes the execution and returns the execution result, the MainTx can continue to execute, and the subsequent executions are the same.

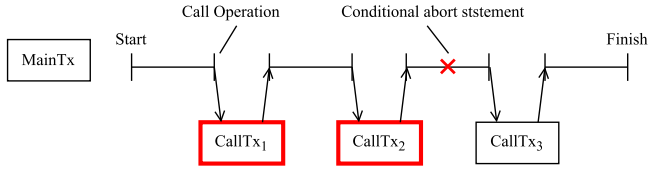


Fig. 5. Execution process of the decomposed transaction. The MainTx is corresponding to the smart contract that is directly called by the original transaction while the CallTx is corresponding to the smart contract that is called by the MainTx's corresponding smart contract.

Note that the transaction decomposition in our SC-CHEF follows the same assumption in [18] and [24] that the invocation logic between smart contracts accessed by each transaction is known in advance. Since smart contracts can be invoked against each other, the invocation logic between smart contracts within a transaction is unpredictable. With the prior knowledge of the invocation logic between smart contracts within each transaction, each transaction are decomposed into multiple subtransactions (at the granularity of smart contracts), which are orchestrated into a directed acyclic diagram (DAG) based on the invocation logic between smart contracts. Moreover, the execution of a smart contract acts as a function invocation, and its execution depends on the execution result (return value) of the previous smart contract, which serves as its input parameters [25]. Therefore a subtransaction must be executed with the execution results of its predecessor before it can be executed. In particular, the invocation logic between smart contracts within a transaction can be obtained by preexecution during the transaction construction phase. Even if a malicious client forges a transaction with incorrect read/write set, the nodes in SC-CHEF will detect during the transaction execution phase that the actual smart contracts accessed by the transaction differ from the predeclared ones, and such a transaction will be rolled back and executed serially with the transactions in its read–write set. Note that SC-CHEF nodes are able to coexist in the same blockchain system with regular blockchain nodes, because they only differ in the execution of smart contract transactions, and the underlying consensus mechanism remains the same.

In addition, smart contracts support active rollback triggered by the logic, which may cause cascaded rollbacks [26]. This problem can be solved by utilizing the idea of "early write visibility" proposed by Faleiro et al. [21]. Therefore, as shown in Fig. 5, CallTxs before the abort statement are marked as abortable to

prevent their abortable status to be read by other transactions before the entire transaction is guaranteed to commit. The execution details of the abortable CallTx will be introduced later.

Unlike concurrency control systems, such as OCC and 2PL, where each thread processes transactions one by one, SC-CHEF processes transactions in batches, which is more suitable for the blockchain system. Therefore, in the transaction decomposition phase, SC-CHEF will take out a batch of transactions from the transaction pool, and then decompose each transaction into one MainTx and multiple CallTxs (both of them are referred to as Tx-piece hereafter) according to the invocation relationship between contracts.

C. Execution Order of Tx-Pieces

In the section above, we have explained the execution order of Tx-pieces, in which a MainTx calls all CallTxs in turn. However, in the concurrent scenario, two Tx-pieces belonging to different transactions may conflict when these two Tx-pieces read or write same variables of the same smart contract.

Assume that there exists read–write, write–read, or write–write conflicts between Tx-piece$_1$ and Tx-piece$_2$, and they belong to transactions Tx$_1$ and Tx$_2$ respectively, where Tx$_1$ is scheduled to commit before Tx$_2$. In order to ensure the serializable scheduling, Tx-piece$_1$ must be executed before Tx-piece$_2$.

D. Partition Dependency Graph Construction

Each smart contract in the blockchain corresponds to a contract account identified by a unique address, and each contract account possesses an independent storage space. In SC-CHEF, all smart contracts in the blockchain are allocated to multiple mutually exclusive partitions according to the address corresponding to their contract account. Each partition corresponds to a single CPU thread. Since the storage space of each smart contract is independent, there is no shared data between these threads.

In the transaction decomposition phase, a batch of transactions is decomposed into multiple MainTxs and CallTxs. These Tx-pieces will be mapped to different partitions according to their corresponding smart contract account addresses. Therefore, a thread will execute Tx-pieces from different transactions. In order to handle the access of multiple Tx-pieces to conflicting data in the partition, the thread of each partition has to create a dependency graph similar to some of deterministic concurrency control algorithms [17] based on the ordering requirement mentioned in Section III-C. The details of the dependency graph construction are shown in Algorithm 2.

E. Partition Concurrency Execution

The partition dependency graph construction phase will output multiple directed acyclic graphs (DAG). SC-CHEF explores a breadth first search algorithm [27] which initialized with multiple root nodes to traverse and execute all nodes in the graph. A node in DAG represents a Tx-piece (MainTx or CallTx). Next, we will introduce under what circumstances a Tx-piece can be executed.

Algorithm 1: Transaction Decomposition.

Input:
The set of Transactions with $AccessList$, Tx_n;
Number of Threads, N;
Output:
N Dependency graph for threads, DG_n;

```
for i = 1 → N do
    Initialize an DG_i;
for each Tx_i ∈ Tx_n do
    Decompose Tx_i into multiple SubTx_n according to Tx_i.AccessList;
    for each SubTx_i ∈ SubTx_n do
        idx = map(SubTx_i.Address);
        UpdateGraph(DG_idx, SubTx_i);
return DG_n;
```

Algorithm 2: UpdateGraph.

Input:
A dependency graph DG;
A sub-transaction, $SubTx$;
Output:
An updated dependency graph DG;

```
1  for each AcceInfo_i ∈ SubTx.AccessList do
2      if AcceInfo_i.Operation == Read then
3          Tx = FindLastWriteTx(AcceInfo_i.Addr, AcceInfo_i.Key);
4          if Tx ≠ null then
5              DG.AddEdge(Tx, SubTx); //w-r conflict
6          else
7              DG.AddEdge(null, SubTx);
8          UpdateLastReadTx(SubTx, AcceInfo_i);
9      else if AcceInfo_i.Operation == Write then
10         Tx = FindLastWriteTx(AcceInfo_i.Addr, AcceInfo_i.Key);
11         if Tx ≠ null then
12             DG.AddEdge(Tx, SubTx); //w-w conflict
13             goto Line 17;
14         TxSet = FindLastReadTxSet(AcceInfo_i.Addr, AcceInfo_i.Key);
15         for each Tx_i ∈ TxSet do
16             DG.AddEdge(Tx_i, SubTx); //r-w conflict
17         UpdateLastWriteTx(SubTx, AcceInfo_i);
18 return DG;
```

In the execution phase, MainTx can be in three states, `Uncompleted`, `Blocking`, and `Completed`. As introduced in Section III-B, there are two types of CallTxs: nonabortable and abortable. The abortable CallTx can be in three states, `Unexecuted`, `Executed`, and `Completed`. The nonabortable CallTx can be in the state of `Unexecuted` or `Completed`.

In addition, each CallTx has two flags `CallingFlag` and `CommitFlag` initialized to false, which, respectively, indicates that its MainTx is waiting for it to return the result and the entire transaction is guaranteed to commit. All MainTxs are initialized as `Uncompleted` and all CallTxs are initialized as `Unexecuted`.

In the process of traversing the DAG, whenever a Tx-piece is taken from the queue of nodes to be executed, the following checks are used to determine whether this Tx-piece can be executed.

If the type of the Tx-piece is MainTx, the thread of the partition performs the following two checks.

1) First, the thread checks whether the MainTx's parent nodes are all `Completed`.
2) Second, the thread checks whether the MainTx is not `Blocking`. The `Blocking` state means that this MainTx is waiting for its CallTx to return the execution result.

If these two checks above are true, the Tx-piece of MainTx starts to execute. During its execution, if it encounters a call statement, this MainTx will store input parameters of the called smart contract and the EVM state in its own cache space and change the corresponding CallTx's `CallingFlag` to true. Then, it will set the state of the MainTx to `Blocking`, and exit the execution. Otherwise, if the Tx-piece of MainTx is successfully executed, it will change its state to `Completed`. Whenever the conditional abort statement is guaranteed not to be executed, the MainTx's corresponding abortable CallTxs' state will be changed to `Completed` from `Executed`. As for its nonabortable CallTxs, their `CommitFlag` will be converted to true.

If the type of the Tx-piece is CallTx, the thread of the partition performs the following three checks.

1) First, the thread checks whether the CallTx's parent nodes are all `Completed`.
2) Second, the thread checks whether the CallTx's `CallingFlag` is true, which means that the corresponding MainTx is waiting the execution result from the CallTx.
3) Third, if the Tx-piece is an abortable CallTx, skip this check. If it is a nonabortable CallTx, the thread checks whether the CallTx's `CommitFlag` is true. This is to check whether the transaction to which the Tx-piece belongs is guaranteed to commit.

When three checks above are true, the CallTx will be executed depends on its type. For the abortable CallTx, it will be executed and changed to `Executed`. For nonabortable CallTx, it will be executed directly. Both of them will return the execution result to their corresponding MainTx after the finishing execution.

In the case that the conditional abort statement is executed, the entire transaction must be rolled back. SC-CHEF triggers all CallTxs to perform the rollback operation through MainTx, which is to roll back to the original snapshot of the smart contract corresponding to each Tx-piece. Due to the third check, these rollbacks will not cause inconsistent reads of other normal transactions.

Intuitive Example: To better describe how smart contract transactions are executed concurrently in SC-CHEF, an intuitive example is shown in Fig. 6. Suppose there are four smart contracts SC_A, SC_B, SC_C, and SC_D, which are mapped to three different partitions, as shown in the figure. Each partition corresponds to a thread. In SC-CHEF, transactions are executed in batches in the form of blocks. Within a period of time, there are four smart contract transactions (Tx_1, Tx_2, Tx_3, and Tx_4) in a block that need to be executed, each of which performs several read or write operations on some of the above smart contracts. For example, the expression Tx_1: w(SC_A), w(SC_B), r(SC_C) indicates that transaction Tx_1 first performs a write operation to SC_A, then a write operation to SC_B, and finally a read operation

Algorithm 3: Partition Concurrency Execution.

Input:
A dependency graph composed of Sub-Transactions, DG;

```
1  //This algorithm is executed concurrently in each thread;
2  //Use breadth first search(BFS) to traverse DG with the help of a queue;
3  Initialize a queue Q;
4  Create an empty root node Root, its child nodes are all the nodes of DG whose in-degree is 0;
5  Q.EnQueue(BFS(Root, DG)); //Visit only one node at a time;
6  while Q ≠ empty do
7      SubTx := Q.DeQueue();
8      if SubTx.Type == MainTx then
9          if SubTx.Parents.State == Completed&& SubTx.State ≠ Blocking then
10             Execute(SubTx);
11             Q.EnQueue(BFS(SubTx, DG));
12         else
13             Q.Enqueue(SubTx);
14     else if SubTx.Type == CallTx then
15         if SubTx.Parents.State == Completed && SubTx.CallingFlag == True then
16             if !SubTx.IsAbortable() then
17                 if SubTx.CommitFlag == True then
18                     Execute(SubTx);
19                     Q.EnQueue(BFS(SubTx, DG));
20                 else
21                     Q.Enqueue(SubTx);
22             else
23                 Execute(SubTx);
24                 Q.EnQueue(BFS(SubTx, DG));
25         else
26             Q.Enqueue(SubTx);
```

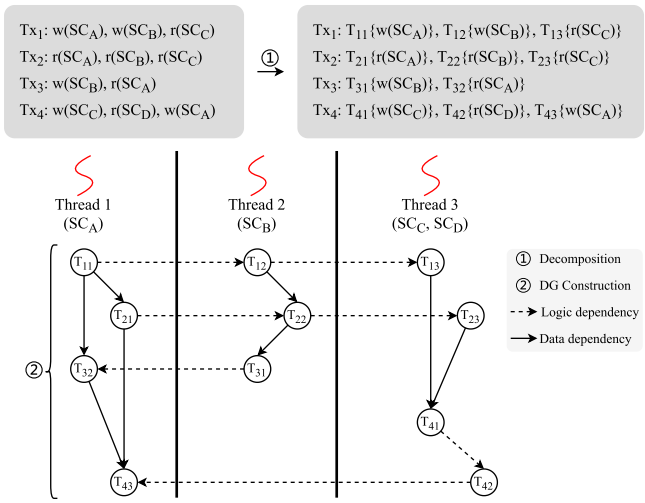


Fig. 6. Example of partition dependency graph (DG) construction.

from SC_C. Each transaction is first decomposed into multiple subtransactions based on its read and write logic, where each subtransaction represents an access operation to a smart contract. Then subtransactions are mapped to the corresponding threads based on the data they access. Each partition thread constructs a local dependency graph (DG) according to the determined order of transactions and the read and write relationships between subtransactions. There are two kinds of dependencies between subtransactions: 1) logic dependency (dashed arrows) between subtransactions belonging to the same transaction and 2) data dependency (solid arrows) between subtransactions that access the same data in the same DG. Only subtransactions with no dependencies can be executed, which is guaranteed by the checks mentioned in Algorithm 3 before executing each subtransaction. A subtransaction can only be executed when its parent nodes have all been executed and the transaction to which the subtransaction belongs is guaranteed to be committed.

F. Execution Optimization of Validation Nodes

SC-CHEF runs on the execution layer of nodes in the blockchain system, which is independent of the consensus layer [28]. Whenever SC-CHEF completes the execution of a batch of transactions and embeds it in a newly generated block, this block will be broadcast by the consensus engine to other verification nodes in the network, where the transactions in the block will be reexecuted. Since the overall sequence of a batch of transactions in the block has been determined, the SC-CHEF running on the execution layer of the verification node can generate the same serializable schedule as the mining node and reach a consistent final state.

Note that, during the transactions' execution phase of the mining node, SC-CHEF does not know which transactions will be aborted, because it is unknown whether the conditional abort statements of their related smart contracts will be executed. Thus, it is necessary to check the commit condition to prevent cascaded abort, as described in Section III-E. However, by the time when the verification node reexecutes the transactions in the block, whether the conditional abort statement in the smart contract is executed can be inferred (because the aborted transactions will not be packed into the block). Therefore, in the verification phase, SC-CHEF cancels the commit condition check of all Tx-pieces, which means that all CallTxs are nonabortable. It can further speed up the execution of transactions in the validation phase.

IV. SECURITY ANALYSIS

SC-CHEF optimizes the performance of the smart contract execution layer of the blockchain and is orthogonal to the consensus layer, which means it does not depend on the underlying consensus algorithm. SC-CHEF can be applied to any blockchain that uses any consensus algorithm, as long as it supports smart contracts. Therefore, SC-CHEF does not need to add additional security assumptions to the original threat model of the blockchain system.

The only requirement of SC-CHEF at the smart contract execution layer is that the read and write sets of transactions are known in advance, which can be declared by the clients that initiated the transactions. However, the clients may be malicious. Next, we demonstrate that the security of the execution of smart contracts can be guaranteed even when this requirement can not be satisfied.

SC-CHEF requires clients to provide the read and write sets of transactions, so as to decompose transactions and obtain deterministic order of subtransactions. Suppose a malicious client tries to initiate a transaction with incorrect read and write sets. This transaction will first be broadcast to the mining nodes, and these nodes will check whether the actual read–write set of this transaction is the same as the declared read–write set during the execution of this transaction. Once an error is detected, this transaction will be aborted immediately.

Once a transaction is rolled back due to an error during execution, its previous modifications to the smart contracts need to be restored. Therefore, SC-CHEF can prevent the modifications caused by the rolled back transactions from being read by other transactions, which ensures serializability of concurrent execution.

V. EVALUATION

In this section, we thoroughly evaluate the performance of our SC-CHEF compared to three other methods, OCC, 2PL, and Serial.

1) *OCC:* The OCC stands for optimistic concurrency control. Since most of the methods currently applied to the concurrent execution of smart contracts are variants of OCC, here we implement the original OCC algorithm based on [29].
2) *2PL:* The 2PL concurrency control system is implemented based on [30]. In order to avoid deadlock, during the growing phase, locks for the smart contract state variables will be applied in lexicographical order before the execution.
3) *Serial:* The Serial method represents the original Ethereum (public version) method of executing transactions, which processes transactions one by one serially.

TABLE I
PARAMETER SETTING

Workload Parameters	**Value**
Total amount of data	100,000
Zipfian parameter θ	$0 \sim 0.9$
Number of read-modify-write operations per transaction	10
Experiment Parameters	**Value**
Maximum number of threads	20
Blocksize (number of transactions)	1024

All these three methods above and our SC-CHEF are implemented in the private Ethereum platform [14].

A. Setup

The experiment in this article runs on four physical server nodes, and each server is consisted of two Intel Xeon Silver CPU 4210. Each CPU possesses 10 cores and 20 threads. Our experiments will first evaluate the performance of the mining node's transaction execution, and then evaluate the performance of a blockchain system composed of four physical nodes.

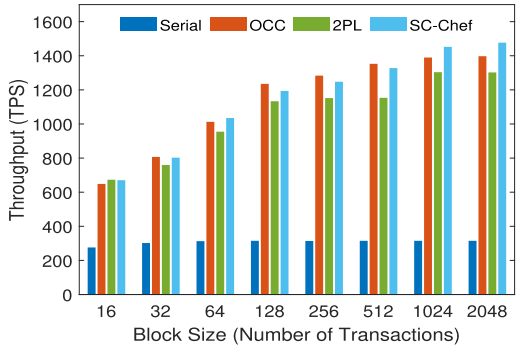
B. Workload

We follow the similar way to develop the evaluation workload as prior work [11], [18], [28]. Our workload includes two kinds of smart contracts: 1) YCSB [31] and 2) TEST (contracts that perform multiple invocations to YCSB contracts). YCSB is implemented in the form of smart contract that stores a number of key-value pairs and provides functional interfaces for reading and writing it. In order to access the data in YCSB smart contracts, multiple TEST smart contracts are deployed, each of which performs ten read–modify–write operations. For complex smart contracts, we insert a quick sort operation between each read and write operation. Note that each read or write operation represents a call to the YCSB smart contract.

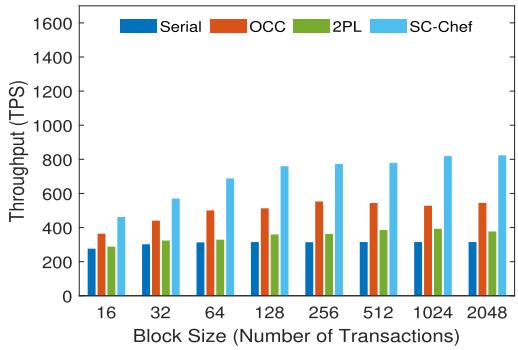
The workload and experiment configuration is shown in Table I. 100 000 data (key-value pairs) are evenly deployed to multiple YCSB smart contracts, the number of which is equal to the number of partitions mentioned in Section III-D. The data access pattern follows the Zipfian distribution [32], and its parameter θ varies from 0 to 0.9. Parameter θ represents the data contention rate. The larger the θ value is, the higher the data contention rate is. The higher the data contention rate is, the more intensive the data is accessed by transactions. In high contention workloads, a large number of transactions access a small amount of hot data in YCSB contracts, which causes frequent data contention between transactions.

C. Effect of Block Size

Block size affects the speed of concurrent execution of transactions. The larger the block size is, the higher the concurrency potential of transactions in a block is. We first evaluate the impact of the block size on the performance of mining nodes. In this experiment, by changing the block size in logarithmic steps from 16 to 2048 when the number of threads is 20, the performance results under different contention workloads are shown in Fig. 7(a) and (b). As shown in these figures, no matter how the block size and contention parameter θ changes, Serial's throughput remains unchanged. This is because the mode of executing transactions serially does not take advantage of multithreading. Although the increase in block size increases the concurrency potential of a batch of transactions, it does not affect the serial method. The throughput of OCC, 2PL, and SC-CHEF increases as the block size increases, and the increase speed gradually slows down. Their throughput does not change significantly when the block size is equal to 1024 or 2048. Therefore, we set the block size to 1024 in the following experiments.



(a)



(b)

Fig. 7. Effect of block size on throughput (transactions per second) under different contention.(a) Low contention ($\theta = 0$). (b) High contention ($\theta = 0.9$)

D. Effect of Contention

Under high contention workloads, our SC-CHEF outperforms OCC and 2PL by 60% and 70% on average, respectively. In the following experiment, we evaluate the effect of contention on the throughput of the mining node in varying number of threads. The contention in our experiment indicates that two transactions access the same variable of the same smart contract. We control the degree of contention by changing the parameter θ of the Zipfian distribution. The larger the θ is, more contention transactions accessing the same set of contract variables cause.

Fig. 8 shows the performance results under the low contention. Serial's throughput does not change with the number of threads, because it does not use multithreaded resources, which is also one of the reasons for the poor performance of Ethereum. When

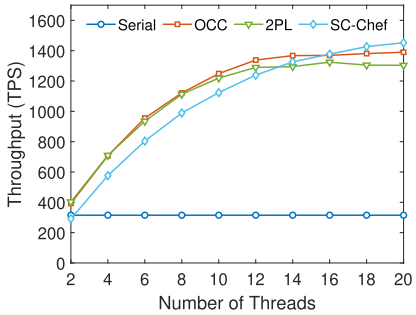


Fig. 8. Effect of number of threads on throughput under low contention ($\theta = 0$).

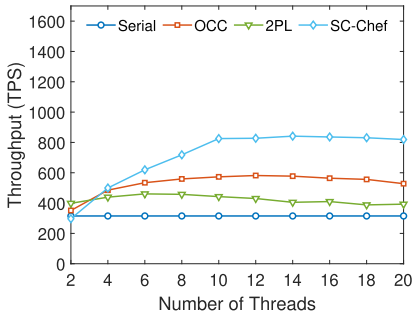


Fig. 9. Effect of number of threads on throughput under high contention ($\theta = 0.9$).

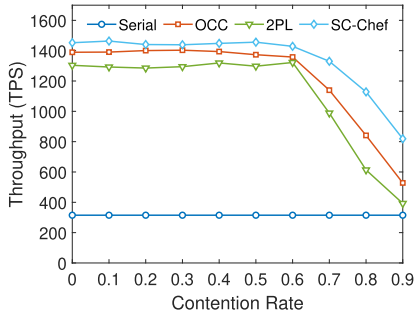


Fig. 10. Effect of contention rate on throughput when the number of threads is 20.

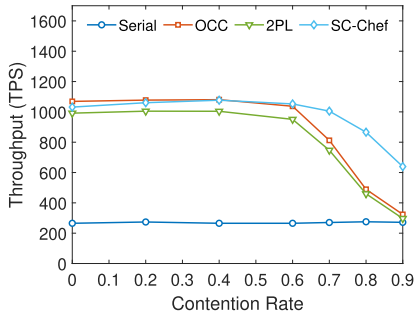


Fig. 11. Throughput of a blockchain system under varying contention.

the number of threads is increasing, the throughput of all of them, SC-CHEF, OCC, and 2PL, is increasing. Specifically, when the number of threads is 20, the throughput of OCC, 2PL, and SC-CHEF is 4.4, 4.1, and 4.6 times that of Serial, respectively. In particular, as the number of threads increases, SC-CHEF gradually outperforms OOC and 2PL. This is because during the commit phase of a transaction in OCC, each thread accesses the same global object that is used for version control. For 2PL, each thread accesses the same object used for lock management when requesting and releasing locks that a transaction accesses. Therefore, as the number of threads increases, the competition for shared resources increases between threads in both OCC and 2PL. In SC-CHEF, each thread executes subtransactions in its own partition and each partition maintains a disjoint set of smart contracts, so there is less competition for shared resources between threads, which shows that our SC-CHEF performs better.

Fig. 9 shows the performance results under the high contention workloads. Like Serial, with the increasing number of threads, the performance of OCC does not change significantly, which is only 1.6 times that of Serial. Similarly, the throughput of 2PL is only 1.2 times that of Serial. In the case of high contention, SC-CHEF can still maintain an upward trend. When the number of threads is 20, SC-CHEF's throughput is 2.6 times that of Serial. In the case of the high contention, in addition to competition for shared resources, transactions in OCC will frequently be reexecuted due to the modification of conflicting data, which severely deteriorates the performance. For 2PL, high contention will increase the time it takes for transactions to apply for locks. Threads of SC-CHEF have determined the execution order of each Tx-piece before they start executing the Tx-pieces, so there will be no reexecution and lock application in SC-CHEF.

We evaluate the performance of these four methods when the number of threads is 20 and the parameter θ of the Zipfian distribution varies from 0 to 0.9. As shown in Fig. 10, the throughput of OCC, 2PL, and SC-CHEF starts to decrease when θ is greater than 0.6. The difference is that with the increase of θ, OCC and 2PL decreases faster, which shows that SC-CHEF is more able to withstand varying contention workloads.

Furthermore, we evaluate the performance of our SC-CHEF in a blockchain system consisted of multiple nodes. Note that OCC and 2PL are not deterministic concurrency control algorithms, it is necessary to design deterministic concurrency algorithms for verification nodes. For the concurrent verification phase of OCC and 2PL, we implement a deterministic concurrent execution algorithm based on the fork-join method [8] with the help of the directed acyclic graph generated by the mining node. We choose the proof-of-authority (PoA) [33] consensus algorithm provided by Ethereum specifically for permissioned blockchain, which can specify mining nodes. In this experiment, we designate one node as a mining node, and the remaining three nodes as verification nodes.

We perform this experiment by running this system until ten new blocks are generated, which means that 10 240 transactions are successfully executed and synchronized to all nodes. The overall time spent is defined as the time when the mining node starts to generate new blocks to the time the last block is successfully synchronized to all nodes. In Fig. 11, we can see that the overall experimental results are similar to the results of a single node, as shown in Fig. 10. When the contention rate is higher than 0.6, both SC-CHEF, 2PL, and OCC's throughput begin to decline.

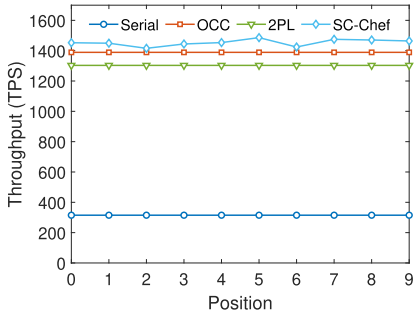


Fig. 12. Effect of the position of the conditional abort statement on throughput under low contention ($\theta = 0$).

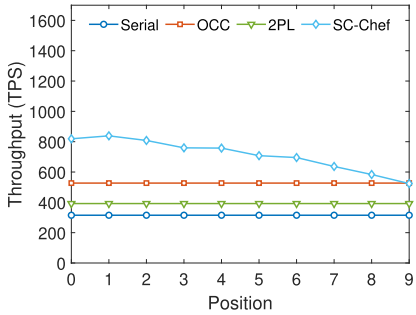


Fig. 13. Effect of the position of the conditional abort statement on throughput under high contention ($\theta = 0.9$).

However, the performance of SC-CHEF is always better than OCC and 2PL. Especially in the case of $\theta = 0.9$, SC-CHEF outperforms OCC and 2PL by 98% and 115%, respectively. Compared with the performance of a single node, the performance of the above methods is slightly reduced because of the communication and verification overhead between multiple nodes.

E. Effect of Conditional Abort Statement

Under low contention workloads, the location of conditional abort statement (CAS) has little effect on the performance of SC-CHEF. Under high contention workloads, the later CAS appears in the execution process of smart contracts, the lower the throughput of SC-CHEF is, but our SC-CHEF still outperforms OCC and 2PL. As mentioned in Section III-E, the CAS in the smart contract means that the smart contract will actively roll back when certain conditions are met, which affects whether the decomposed Tx-piece can be executed. We evaluate the effect of the position of the CAS in the smart contract on throughput. We define the position of the CAS as the number of read–modify–write operations before the CAS in a contract. For example, if the position is equal to 2, it means that the contract performs a CAS check after completing two read–modify–write operations to determine whether the execution will be rolled back. Since the TEST smart contract we deployed performs ten read–modify–write operations, the position of the CAS varies from 0 to 9 in this experiment with a step size of 1.

Experimental results are shown in Figs. 12 and 13. We can observe that the throughput of OCC, 2PL, and Serial remains unchanged as the position of the CAS changes because they simply treat every transaction as a unit. For SC-CHEF, its throughput is hardly affected by the position of CAS under low contention conditions. In the case of high contention, SC-CHEF's throughput, which is still better than that of OCC, 2PL, and Serial, decreases as the position of CAS moves backward. In mining nodes, the position of the CAS in a transaction can be anywhere in a smart contract. Since it is unknown whether CAS will be executed during the mining phase, nodes must check whether the CAS of each transaction is executed to determine whether the transaction will be aborted. However, in verification phase, all transactions in a block are guaranteed to be committed. Therefore, when verification nodes reexecute the transactions in a block with SC-CHEF, they do not need to consider the impact of the CAS.

VI. DISCUSSION

This section discusses the limitations, potential improvements, and applications of the proposed SC-CHEF.

Limitation: The transaction decomposition step of SC-CHEF requires prior knowledge of the read/write sets of transactions from the clients, which gives clients the opportunity to conduct denial-of-service [34] attacks. Therefore, the privileges of the clients should be restricted.

Improvement: Automated analysis in smart contract bytecode level [35] could explore finer-grained concurrency and detect unknown smart contract invocations in advance to prevent excessive transaction rollback. The mapping of smart contract addresses on threads can be unbalanced, which can lead to underutilization of multithreaded resources. Some load balancing mechanisms [36] can be investigated to optimize multithreaded resource allocation for smart contracts in SC-CHEF.

Application: Our SC-CHEF can be used to optimize blockchain applications with complex computation and high contention workloads. For example, SC-CHEF is suitable for industrial IoT applications with a large number of distributed nodes and complex computation.

Influence of Mining Nodes: As with regular blockchain systems, the mining nodes in the proposed system have the right to decide the execution order of smart contract transactions. In each generation period of a new block, each mining node executes a batch of transactions concurrently and records them into a new block. Each mining node has the right to prioritize the execution of transactions with high transaction fees, while transactions with low fees will be executed in later blocks. By leveraging the multicore CPU resources of the hardware, our system optimizes the efficiency of concurrent execution of smart contract transactions for each mining node. When the overhead of executing complex smart contract transactions is reduced, mining nodes can execute more transactions in a single block, thus reducing the impact of miners controlling the execution order of transactions. Namely, when transactions can be executed more quickly, clients do not need to compete for higher transaction execution priority by increasing transaction fees.

VII. RELATED WORK

In this section we review related research works on smart contract related to this article.

Smart Contracts Concurrency: Dickerson et al. [8] enabled smart contracts to be executed in parallel in order to improve their efficiency with the help of 2PL concurrency control system. Sharma et al. [11] proposed Fabric++, which speeds up the execution of Hyperledger's smart contracts by reordering and early aborting transactions to increase the number of successfully executed transactions per unit time. Since the basic execution structure of the smart contract transaction of Hyperledger is completely different from that of Ethereum, the Fabric++ algorithm cannot be applied to Ethereum or blockchains similar to Ethereum. Pang et al. [10] proposed a method for concurrent execution of smart contracts based on the combination of OCC variants and batch processing features, which can speed up the execution of smart contracts on mining nodes and verification nodes. Reijsbergen et al. [19] proposed an analytical model for estimating transaction execution acceleration with a given amount of concurrency, and used this model to analyze the concurrency potential of seven different blockchains. Anjana et al. [9] proposed a method of applying BTO and MVTO to the concurrent execution of smart contracts, which also accelerates the contract execution speed in the verification phase with the help of DAG generated in the mining phase. Yu et al. [37] proposed a smart contract execution structure whose block are generated concurrently. Most of these methods are performance optimizations for the execution of smart contracts on permissionless blockchain. Moreover, these smart contract concurrency control systems are designed in combination with variants of OCC or 2PL, which simply treat transactions as atomic units for concurrency scheduling, so they perform poorly in high contention workload scenarios. More importantly, in order to evaluate the performance of their algorithms, all of these Ethereum-based work only simulated the execution of smart contracts in a Java or C++ simulation environment, which is quite different from actual smart contracts running in the real blockchain system. Therefore, it is difficult for their experiments to reflect the actual effects of their algorithms in real blockchain systems. Some research work added concurrency to the execution of smart contracts through sharding [38] and off-chain mechanisms [18], [39]. The sharding and off-chain mechanisms spread the original on-chain computation to multiple on-chain computation shards or off-chain computation entities, while smart contract transactions are executed in each computation node. Note that our SC-CHEF optimizes the transaction execution within a single computation node, which is orthogonal to the underlying consensus mechanism. Therefore our SC-CHEF is compatible with the consensus layer mechanisms like sharding and off-chain.

Characterization of Smart Contracts: Current characterization of smart contracts were studied mostly from three aspects, design patterns, the programming code, and execution traces. Design patterns of various smart contracts on different blockchain platforms are well studied [40], and implementation details of such contracts are also explored. Chen et al. [41] proposed to use the techniques of graph analysis to extract the money flow patterns of smart contracts in public Ethereum, and then detect their abnormal behavior [42]. Trace vulnerabilities from one million smart contracts were characterized in detail, and a practical tool based on the implications of such study were proposed to test trace properties [43]. In order to let the public better understand smart contracts, ontologies were presented to reduce the conceptual ambiguity of smart contracts [44]. In addition, one interesting finding is that most contracts are copies of other contracts based on the analysis of contract equality and similarity [7]. Wöhrer et al. [5] emphasized that for the maintenance of blockchain projects, the data and logic in decentralized applications need to be deployed separately to prevent entangled coupling, so the call operations between smart contracts are bound to be frequent. Zhang et al. [6] applied software patterns to the design of complex decentralized smart contract applications to achieve efficient interoperability. Due to the modularization of functions, there are complex calling relationships between smart contracts. Kiffer et al. [7] examined how smart contracts on Ethereum are created and how contracts interact with each other. By analyzing the contract interaction graph, they found that most of the smart contract execution is triggered by another smart contract, which shows that it is common for contracts to call other contracts.

Smart Contracts Security and Privacy: Security and privacy of smart contracts have attracted attention from the research community. Security analysis of smart contracts was conducted, and vulnerabilities that can be used to manipulate the smart contracts to gain illegal profit were discovered [45]. Surprisingly, smart contracts could be abused for criminal purposes. This indicates our community has to promote clear administrative procedures to regulate the development ecosystem of smart contracts [46]. Proactive approaches were first proposed. For example, Bhargavan et al. [47] proposed a formal verification framework to analyze and verify both the runtime safety and functional correctness of smart contracts. With the security design in mind, Mavridou et al. [48] proposed a finite state machine-based approach to design secure smart contracts and a list of security design patterns were also introduced to enhance security and functionality. To secure interactions between smart contracts and external data sources, Zhang et al. [49] proposed an authenticated data feed for smart contracts. For the open nature of permissionless blockchains, privacy is also an important concern. Kosba et al. [50] proposed a tool Hawk to preserve the privacy of smart contracts by enabling people to write the private smart contract without the burden to implement the cryptography. Other security issues were also investigated. For example, Velner et al. [51] demonstrated that using the smart contract can withhold the blocks, which would undermine the entire pooled mining model.

VIII. CONCLUSION

In this article, we have proposed SC-CHEF, a new concurrency control system of smart contract execution based on the transaction decomposition to turboboost smart contract concurrent execution for high content workloads in the permissioned blockchain. Our SC-CHEF could decompose the original transaction into multiple subtransactions that can be executed independently while still respecting the original logic dependency. We have implemented our SC-CHEF in the private Ethereum

platform. Our extensive experiments have shown that our SC-CHEF outperforms the classic OCC and 2PL by 60% and 70% on average in the case of high contention, respectively.

ACKNOWLEDGMENT

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] G. Wood et al., "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[2] N. Szabo, "The idea of smart contracts," *Nick Szabo's Papers and Concise Tut.*, vol. 6, 1997.

[3] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos, "Performance modeling of PBFT consensus process for permissioned blockchain network (hyperledger fabric)," in *Proc. IEEE 36th Symp. Reliable Distrib. Syst.*, 2017, pp. 253–255.

[4] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 3–16.

[5] M. Wöhrer and U. Zdun, "Design patterns for smart contracts in the Ethereum ecosystem," in *Proc. IEEE Int. Conf. Blockchain*, 2018, pp. 1513–1520.

[6] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, "Applying software patterns to address interoperability in blockchain-based healthcare apps," 2017, *arXiv:1706.03700*.

[7] L. Kiffer, D. Levin, and A. Mislove, "Analyzing Ethereum's contract topology," in *Proc. ACM Internet Meas. Conf.*, 2018, pp. 494–499.

[8] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2017, pp. 303–312.

[9] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for concurrent execution of smart contracts," 2018, *arXiv:1809.01326*.

[10] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 11, pp. 5070–5083, Nov. 2022.

[11] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of hyperledger fabric," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2019, pp. 105–122.

[12] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1629–1642.

[13] S. H. Son and R. David, "Design and analysis of a secure two-phase locking protocol," in *Proc. 18th Annu. Int. Comput. Softw. Appl. Conf.*, 1994, pp. 374–379.

[14] "Private Ethereum for Enterprise," 2020. [Online]. Available: https://ethereum.org/en/enterprise/private-ethereum/

[15] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019.

[16] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core OLTP under high contention," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1583–1598.

[17] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *VLDB Endowment*, vol. 8, no. 11, pp. 1190–1201, 2015.

[18] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, and S. Capkun, "ACE: Asynchronous and concurrent execution of complex smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 587–600.

[19] D. Reijsbergen and T. T. Anh Dinh, "On exploiting transaction concurrency to speed up blockchains," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 1044–1054.

[20] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 1–12.

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

[22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*, vol. 7, Reading, MA, USA: Addison-Wesley, 1986.

[23] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 7, pp. 1366–1385, Jul. 2018.

[24] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1337–1347.

[25] T. Chen et al., "SigRec: Automatic recovery of function signatures in smart contracts," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 3066–3086, Aug. 2021.

[26] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, vol. 370, Reading, MA, USA: Addison-Wesley, 1987.

[27] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–10.

[28] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 1085–1100.

[29] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[30] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[32] C. Tullo and J. Hurford, "Modelling Zipfian distributions in language," in *Proc. Lang. Evol. Comput. Workshop/Course ESSLLI*, 2003, pp. 62–75.

[33] A. H. Lone and R. N. Mir, "Reputation driven dynamic access control framework for IoT atop PoA Ethereum blockchain," *IACR Cryptol. ePrint Arch.*, vol. 2020, 2020.

[34] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, "Inferring internet denial-of-service activity," *ACM Trans. Comput. Syst.*, vol. 24, no. 2, pp. 115–139, 2006.

[35] T. Chen et al., "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1503–1520.

[36] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: Optimal transactions placement for scalable blockchain sharding," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 525–535.

[37] L. Yu, W.-T. Tsai, G. Li, Y. Yao, C. Hu, and E. Deng, "Smart-contract execution with concurrent block building," in *Proc. IEEE Symp. Service-Oriented Syst. Eng.*, 2017, pp. 160–167.

[38] C. Luo et al., "Fission: Autonomous, scalable sharding for IoT blockchain," in *Proc. IEEE 46th Annu. Comput., Softw., Appl. Conf.*, 2022, pp. 956–965.

[39] Y. Wang et al., "iBatch: Saving Ethereum fees via secure and cost-effective batching of smart-contract invocations," in *Proc. ACM 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2021, pp. 566–577.

[40] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," in *Proc. Int. Workshops, WAHC, BITCOIN, VOTING, WTSC, TA*, 2017, pp. 494–509.

[41] T. Chen et al., "Understanding Ethereum via graph analysis," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2018, pp. 1484–1492.

[42] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *Proc. Financial Cryptography Data Secur.: FC Int. Workshops, WAHC, BITCOIN, VOTING, WTSC, TA*, 2017, pp. 478–493.

[43] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.

[44] J. de Kruijff and H. Weigand, "Ontologies for commitment-based smart contracts," in *Proc. OTM Conf.: Confederated Int. Conf.: CoopIS, CTC, ODBASE*, 2017, pp. 383–398.

[45] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[46] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Investigating the future of criminal smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 283–295.
[47] K. Bhargavan et al., "Formal verification of smart contracts," in *Proc. ACM Proc. Workshop Program. Lang. Anal. Secur.*, 2016, pp. 91–96.
[48] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," in *Proc. Financial Cryptography Data Secur.: 22nd Int. Conf.*, 2018, pp. 523–540.
[49] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 270–282.
[50] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 839–858.
[51] Y. Velner, J. Teutsch, and L. Luu, "Smart contracts make bitcoin mining pools vulnerable," in *Proc. Financial Cryptography Data Secur.: FC Int. Workshops, WAHC, BITCOIN, VOTING, WTSC, TA*, 2017, pp. 298–316.

Dibo Xian received the B.Sc. degree in electronic science and technology from Northeastern University, Shenyang, China, in 2019. He is currently working toward the M.Sc. degree in electronic science and technology with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen.

His research interests include blockchain and smart contracts.

Xuetao Wei (Member, IEEE) received the Ph.D. degree in computer science from the University of California, Riverside, CA, USA, in 2013.

Since 2019, he has been an Associate Professor with the Southern University of Science and Technology, Shenzhen, China. He was an Assistant Professor and then promoted to an Associate Professor with the University of Cincinnati, OH, USA. His research interests include industrial metaverse, blockchain, and Internet of Things.