mathematics

Article

Adaptive Parallel Scheduling Scheme for Smart Contract

Wenjin Yang [1,2], Meng Ao [2], Jing Sun [3], Guoan Wang [1], Yongxuan Li [1], Chunhai Li [4,*] and Zhuguang Shao [2,*]

1 School of Cyberspace Science & Technology, Beijing Institute of Technology, Beijing 100081, China; wenjinyang@bit.edu.cn (W.Y.); dylanwang@bit.edu.cn (G.W.); liyongxuan@bit.edu.cn (Y.L.)
2 Tencent Inc., Shenzhen 518055, China; mengao@tencent.com
3 School of Computer Science, University of Auckland, Auckland 1023, New Zealand; jing.sun@auckland.ac.nz
4 Guangxi Engineering Research Center of Industrial Internet Security and Blockchain, Guilin University of Electronic Technology, Guilin 541004, China
* Correspondence: chunhaili@guet.edu.cn (C.L.); leonzgshao@tencent.com (Z.S.)

Abstract: With the increasing demand for decentralized systems and the widespread usage of blockchain, low throughput and high latency have become the biggest stumbling blocks in the development of blockchain systems. This problem seriously hinders the expansion of blockchain and its application in production. Most existing smart contract scheduling solutions use static feature analysis to prevent contract conflicts during parallel execution. However, the conflicts between transactions are complex; static feature analysis is not accurate enough. In this paper, we first build the dependency between smart contracts by analyzing the features. After numerous experiments, we propose a conflict model to adjust the relationship between threads and conflict to achieve high throughput and low latency. Based on these works, we propose adaptive parallel scheduling for smart contracts on the blockchain. Our adaptive parallel scheduling can distinguish conflicts between smart contracts and dynamically adjust the execution strategy of smart contracts based on the conflict factors we define. We implement our scheme on ChainMaker, one of the most popular open-source permissioned blockchains, and build experiments to verify our solution. Regarding latency, our solution demonstrates remarkable efficiency compared with the fully parallel scheme, particularly in high-conflict transaction scenarios, where our solution achieves latency levels just one-twentieth of the fully parallel scheme. Regarding throughput, our solution significantly outperforms the fully parallel scheme, achieving 30 times higher throughput in high-conflict transaction scenarios. These results highlight the superior performance and effectiveness of our solution in addressing latency and throughput challenges, particularly in environments with high transaction conflicts.

Keywords: parallel scheduling; smart contract; blockchain

MSC: 68M14

check for updates

Citation: Yang, W.; Ao, M.; Sun, J.; Wang, G.; Li, Y.; Li, C.; Shao, Z. Adaptive Parallel Scheduling Scheme for Smart Contract. *Mathematics* **2024**, *12*, 1347. https://doi.org/10.3390/math12091347

Academic Editor: Antanas Cenys

Received: 24 March 2024
Revised: 24 April 2024
Accepted: 25 April 2024
Published: 29 April 2024

1. Introduction

The rapid advancement of blockchain technology has led to an increasing number of individuals preferring to construct their projects on this decentralized system [1]. The way of creating services on the blockchain involves the usage of smart contracts. The concept of smart contract was first proposed by Nick Szabo [2] in the early 1990s. A smart contract is a self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code. The code and the protocols contained are deployed onto the blockchain.

It is also essential to mention that blockchain technology still faces numerous challenges. One of the most critical issues is its low performance and latency [3]. The widespread utilization of blockchain (such as non-fungible token NFT [4] and cryptocurrency [5]) has resulted in the deployment and numerous invocations of smart contracts on this decentralized platform. This high demand has exacerbated the situation, causing blockchain systems to become slower and more burdensome. Consequently, the low performance and the high latency of smart contracts may scarcely enhance the transaction processing speed on the blockchain and limit the scope of its implementation.

Throughput and latency reflect the number of transactions a block can contain and the time it takes for a block to be finalized, which heavily depends on the consensus and smart contract execution strategy applied. Many high-performance schemes, such as DumboBFT [6], Phantasm [7], and HCA [8,9], aim to expand the throughput and reduce the finalization time via consensus mechanisms. Although these consensus algorithms greatly improve the throughput and latency of a blockchain system, conflicts between smart contracts and the transaction re-execution still impact performance, especially with the deployment of numerous smart contracts. To address the issue of smart contract execution, scholars have proposed various creative schemes. Dickerson et al. [10] proposed a new method for miners and validators for executing smart contracts in parallel based on software transactional memory technology. Singh et al. [11] proposed a technique for executing smart contract transactions concurrently. Yu et al. [12] proposed a parallel smart contract model, which improves performance through multi-thread processing of the smart contracts that have no shared variables. Bartoletti et al. [13] proposed a static approximation of swappability based on a static analysis of the sets of keys read/written by transactions. Most proposed methods use static analysis to find the conflict between smart contracts. It is challenging when dynamic cross-contract invocations occur. Moreover, there is randomness and complexity in the process of cross-contract calls, and the results of static analysis may lead to high rollback rates and low parallelism in actual scenarios. Facing these challenges, after researching related work, we propose three research questions that we focus on.

RQ1 **How to solve cross-contracts calling conflicts in parallel execution?** During parallel execution, conflicts can easily occur due to cross-contract calls, necessitating the re-execution of conflicting transactions, which can lower overall efficiency or even result in worse performance than serial execution. Simply employing parallel execution is not sufficient, and adjustments to the execution process are necessary.

RQ2 **How to dynamically adjust the scheduling of smart contracts to achieve parallel execution?** Static analysis of smart contracts is challenging when dynamic cross-contract invocations occur since the contract access patterns are not predictable [14]. The conflict between transactions is complex, and static analysis from smart contracts is not enough. The execution strategy based on static analysis and dynamically adjusted during execution is the most efficient solution.

RQ3 **How to improve the implementation of parallel scheduling to achieve efficient usage of space and time?** When implementing parallel scheduling in permissioned blockchains like ChainMaker, we have found it necessary to optimize the data structure for efficiency. It is crucial to consider how to better implement this technology in engineering practice.

To answer these research questions, in this paper, we propose adaptive parallel scheduling for smart contracts. Our scheme analyzes the characteristics of smart contracts in fields and the setting of smart contracts on ChainMaker to build a contract dependency based on the characteristics in an actual deployment. During the actual execution of transactions, moreover, regarding conflicts between transactions and the number of parallel threads, we propose a conflict model that can be used in adaptive execution. What is more, we propose a conflict factor as an indicator to describe the degree of conflict between contracts. During the transaction process, we adaptively update the conflict factors and classify strong conflict contracts and weak conflict contracts by a conflict threshold. For strong conflict contracts, we use serial execution. For weak conflict contracts, we still use parallel execution. According to this, we build and update the transaction DAG and execute the transactions. After the execution, we build the transaction execution processing to update the execution strategy. In the experiment, we proposed two indicators, transaction parallelism score and transaction parallelism rate, to build a more detailed evaluation of transaction parallelism. What is more, we use bitmap, which is more efficient in both space and time to optimize our schemes in the experiment. We implement our scheme on ChainMaker [15], and the results show that our scheme meets our expectations.

This paper makes the following contributions.

- **Smart contract dependency DAG from static analysis.** By analyzing the relationship of the field characteristics of smart contracts, we build the contract dependency by constructing a contract dependency DAG.
- **Conflict model from numerous experiments.** We build a conflict model that describes the relationship between the conflicts of transactions and the number of parallel threads during the actual execution of transactions. Through a large number of experiments, we construct the distribution of the number of threads when TPS is optimal.
- **Dynamically parallel scheduling.** We propose an adaptive smart parallel scheduling that adapts the conflict factors and dynamically adjusts the serial and parallel scheduling of contracts based on conflict factors.
- **Technical optimization in actual implementation.** We use a bitmap to improve the usage of space and time, and we implement this scheme on ChainMaker and obtain the expected results.

This paper is outlined as follows: In Section 2, we review the backgrounds of the blockchain, smart contracts, and previous related works. In Section 3, we define some indicators and structs we use. In Section 4, we introduce our scheme in detail. In Section 5, we implement our scheme on ChainMaker and analyze the results. In Section 6, we give a conclusion about this paper.

2. Related Work

The throughput rate of the blockchain is mainly subject to the following two aspects: one is the execution of the consensus mechanism, and another is the execution of smart contracts. The improvement of the consensus mechanism increases the throughput rate of the blockchain, like DumboBFT [6], Phantasm [7], and HCA [8,9]. However, with the continuous enrichment of smart contract content and the increasing number of transactions calling smart contracts, only using a better consensus mechanism is not enough. The execution of a transaction is generally the most time-consuming phase of the entire block-processing workflow. Therefore, the execution of smart contracts is becoming another key bottleneck [16], especially the mechanism of the serial execution of smart contract transactions, which greatly limits the throughput rate of the blockchain system.

Inspired by the scheme of the database system, most of the existing solutions on parallel scheduling in smart contracts apply a directed acyclic graph (DAG) [17] to blockchain to avoid the conflict of competitive transactions.

Dickerson et al. [10] proposed that nodes need to classify smart contracts into those that can be processed in parallel and those that can only perform serial computation by using the software transaction memory method. In particular, miners execute transactions in one block in parallel, using abstract locks and inverse logs to dynamically discover conflicts and to recover from inconsistent states.

Singh et al. [11] proposed a technique to execute smart contract transactions concurrently. The scheme is achieved by employing optimistic software transactional memory systems (STMs), primarily utilized by miners. In their scheme, a miner constructs a block comprising a set of transactions, a conflict graph, the hash of the previous block, and the final state of each shared data object. Furthermore, the study introduces a concurrent validator that is designed to re-execute the same smart contracts in the same manner in parallel.

Yu et al. [12] proposed a parallel smart contract model. This scheme uses multithread technology to implement the proposed model, where transactions are executed in parallel. Then they proposed a transaction splitting algorithm to resolve the synchronization problem. Their scheme still has many problems to be solved. They do not have a functional way to obtain shared variables in each transaction. Not all smart contracts with shared variables will conflict when called by transactions. Only judging the conflicts between smart contracts by shared variables is not accurate.

Bartoletti et al. [13] proposed a static approximation of swappability based on a static analysis of the sets of keys read/written by transactions. Then they put forth a strategy for miners to execute multiple smart contracts concurrently. Their technique involves employing parallel computation once the node has calculated the order in which the smart contracts will be executed beforehand.

Jin et al. [18] proposed a two-phase concurrency control protocol to optimize both phases. The primary executes transactions in parallel and generates a transaction dependency graph with high parallelism for validators. Then, a graph partition algorithm is devised to divide the original graph into several sub-graphs to preserve parallelism and reduce communication costs.

Piduguralla et al. [19] implemented a directed acyclic graph (DAG)-based parallel scheduler framework in Hyperledger Sawtooth V1.2.6 for the parallel execution of smart contract transactions. This framework represents transaction dependencies in blocks through a parallel DAG data structure, which helps in throughput optimization.

Relevant prior works are based on preset conditions and assumptions to execute the smart contracts in parallel. Static analysis lacks flexibility, and over time, static analysis approaches may require periodic updates to adapt to new conditions and data. Facing this problem, we use an adaptive smart contract model to dynamically adjust the serial and parallel scheduling of contracts in our scheme. By deploying and conducting experimental verification on ChainMaker v3.0.0, our scheme can effectively improve the parallelism of transaction execution and has performance advantages in throughput and latency.

3. Preliminaries

In this section, we briefly introduce the current blockchain and the key feature field of smart contracts. Namespace, the key field in the setting of smart contracts, can be used to find conflicts between smart contracts. What is more, we introduce the read–write set, which is the most important indicator to verify the execution results of the contract.

3.1. Blockchain and Smart Contracts

Blockchain technology, characterized by its decentralized nature, has significantly enhanced the functionality and applications of smart contracts. Blockchain can be mainly divided into two types: permissionless blockchain and permissioned blockchain [20]. A permissionless blockchain is a type of blockchain that allows anyone to participate in the network without requiring explicit permission or approval from a central authority. A permissioned blockchain is a type of blockchain where access to the network is restricted and controlled by a central authority or a consortium of entities. Unlike permissionless blockchains, where anyone can participate in the consensus process, permissioned blockchains require potential participants to obtain permission to join the network. This can include the ability to read, submit transactions, or participate in the consensus process.

Smart contracts are self-executing contracts with the terms of the agreement directly written into lines of code. They operate on blockchain platforms, ensuring a high level of security and trust in the digital environment. Smart contracts run on blockchain technology, which ensures they are immutable and distributed, meaning once deployed, they cannot be altered and are accessible to all parties involved. There are many feature fields of smart contracts. We introduce the two most important fields: namespace and read–write set.

3.2. Namespace

In a blockchain system, each contract typically sets its own namespace, which serves as a basis for distinguishing the data stored in the current contract from that of other contracts. In practice, the contract name is often used as the namespace, so we can regard the namespace as a way to distinguish smart contracts. In a key-value-based blockchain storage system, the NameSpace is often used as a prefix for the key, ensuring that the keys of different contracts do not interfere with each other during actual storage.

As Figure 1 shows, there are two smart contracts on chain1 named 'Save' and 'Transfer'. The storage namespaces of these two contracts are '/chain1/Save/' and '/chain1/Transfer'.

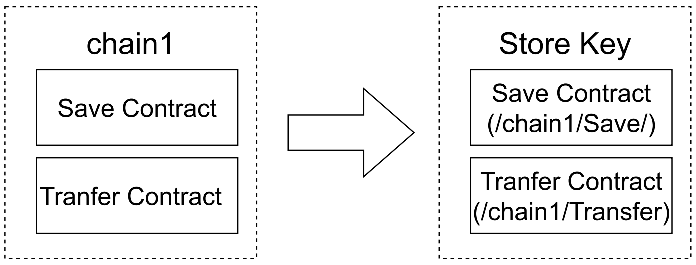

chain1
Save Contract
Tranfer Contract
Store Key
Save Contract (/chain1/Save/)
Tranfer Contract (/chain1/Transfer)

Figure 1. Smart contract and storage namespace.

3.3. Read–Write Set

The read–write set of contracts refers to the set of read-and-write data generated during the execution of a contract, which is abbreviated as RWSet. The structure of the read–write set is shown in Figure 2.

```
// TxRWSet describes all the operations of a transaction on ledger
message TxRWSet {
    // transaction identifier
    string tx_id = 1;
    // read set
    repeated TxRead tx_reads = 2;
    // write set
    repeated TxWrite tx_writes = 3;
}
```

Figure 2. The structure of the read–write set.

In contract operations, each time data are read from the ledger, a read set (ReadSet) is generated. When data need to be written to the ledger, a write set (WriteSet) is generated. ReadWriteSet is closely related to the blockchain ledger storage, and therefore their own Keys also use the contract NameSpace for isolation, which can mask the interference between ReadWriteSets of different contracts.

The structure of ReadSet is shown in Figure 3. ReadSet is mainly used to determine whether there is a data (version) conflict between concurrent transactions after the concurrent execution of contract transactions. If there is a conflict, the dependency between transactions needs to be reflected in the DAG structure, and the execution order of the conflicting transactions needs to be adjusted and re-execute the contract.

```
// TxRead describes a read operation on a key
message TxRead {
    // read key
    bytes key = 1;
    // the value of the key
    bytes value = 2;
    // contract name, used in cross-contract calls
    // set to null if only the contract in transaction request is called
    string contract_name = 3;
    // read key version
    KeyVersion version = 4;
}
```

Figure 3. The structure of ReadSet.

The structure of WriteSet is shown in Figure 4. After the transaction is executed normally, if the world state changes, WriteSet will record detailed change records.

```
// TxRead describes a write/delete operation on a key
message TxWrite {
    // write key
    bytes key = 1;
    // write value
    bytes value = 2;
    // contract name, used in cross-contract calls
    // set to null if only the contract in transaction request is called
    string contract_name = 3;
}
```

Figure 4. The structure of WriteSet.

4. Design

In this section, we first introduce the system model of our scheme. Before proposing the scheme, we introduce some necessary components, such as DAG-based structs and conflict factors, that we define. Then we present our scheme and the algorithms we use.

4.1. System Model

We focus on permissioned blockchain systems such as ChainMaker. Our scheme has six parts: smart contracts, transaction pools, dependency build process, transaction DAG build process, transaction parallel execution, and transaction execution process. Figure 5 shows the system model and workflow. First, we build the dependency DAG of smart contracts based on the relationship between cross-contract calls. Then, we build the transaction DAG and parallel execute the transactions. After the execution, the execution processing part checks the executed transactions and processes the conflicts. Based on the results, we update the dependency DAG and re-execute the conflict transactions.

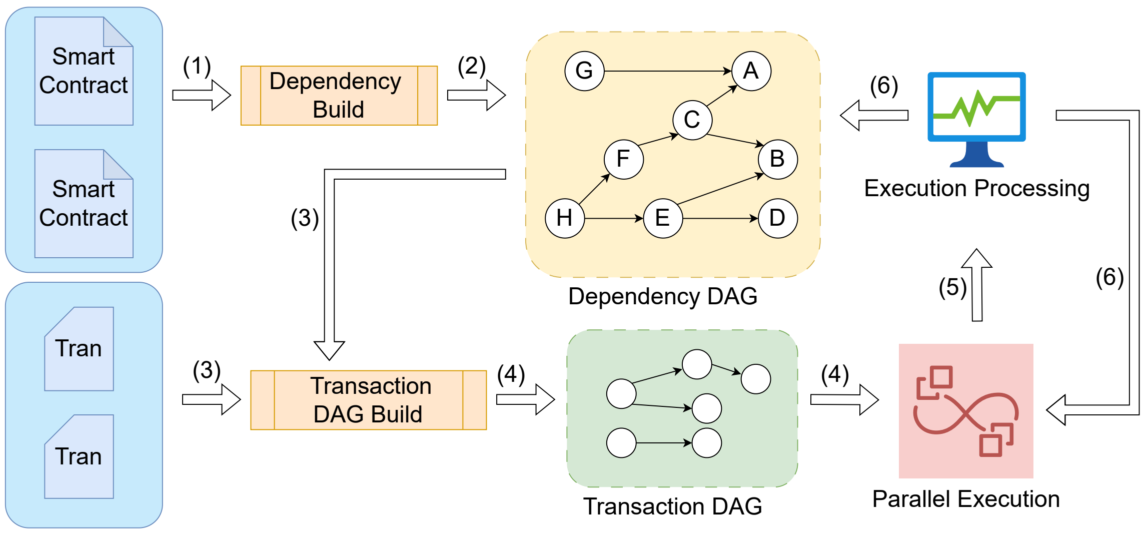


Figure 5. The system model and workflow of our scheme.

4.2. Components

Before we show our scheme, some necessary components should be introduced first.

4.2.1. Contract Dependency DAG

Contract dependency DAG is denoted as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents a set of smart contracts and $\mathcal{E}$ represents a set of relationships, each of which consists of two smart contracts. The relationship means that the contract calls the pointed contract internally. There is only the unidirectional relationship in the contract dependency DAG; i.e., if contract A calls contract B (denoted by $A \to B$), then no relationship exists from contract B to contract A (denoted by $B \to A$).

4.2.2. Transaction DAG

Transaction DAG is denoted as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents a set of transactions that invoke contracts and $\mathcal{E}$ represents a set of orders. From node A to node B (denoted as $A \to B$), the edge represents that transaction B is executed in series after transaction A is completed. This edge indicates the serial execution order between transactions A and B. Like the transaction-dependent DAG graph, each edge of the transaction DAG is also one-way. All the transactions in the same transaction DAG should be serially executed.

4.2.3. Conflict Factors

The conflict between contracts can be described by the conflict factor, which describes the probability of conflict between two contracts and has a range of [0, 1].

We will build a sliding window for two contracts. When the transactions of these contracts conflict, we mark the conflict transaction as 1 and mark the non-conflict transaction as 0. Then we move the sliding window one position to the left, fill in the last position with the token, and update the number of conflicts and the conflict factor (Figure 6).

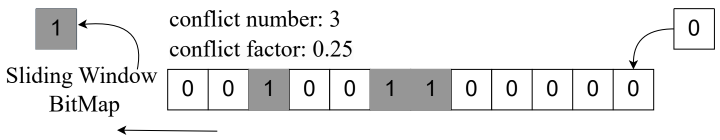


Figure 6. The sliding window to calculate the conflict factor.

For a smart contract, the conflict factor can be used as a characteristic of the contract's interdependence. Contracts with a high conflict factor are defined as strongly dependent contracts and are processed in a conflicting manner when constructing the transaction DAG. Contracts with a low conflict factor are defined as weakly dependent contracts and are processed in a non-conflicting manner when constructing the transaction DAG. Based on the conflict relationship of contract A in the above figure, the threshold value of the conflict factor for strongly dependent contracts is set to 0.6.

4.2.4. Transaction Parallelism Score/Rate

Transaction parallelism score describes the degree of parallelism of transactions in a block.

$$f(n,m) = \begin{cases} n/m & (m < n) \\ 0 & (m = n) \end{cases} \tag{1}$$

In the equation, n represents all the transactions in the block, and m represents the number of transactions in the longest chain of the transaction DAG. When $m = n$, all the transactions in this block should serially execute. For the same number of blocks, the greater the transaction parallelism, the higher the score.

According to the equation of transaction parallelism score, for two blocks containing 100 transactions and 10 transactions, assuming that all transactions in both blocks are parallelizable, their transaction parallelism scores are 100 and 10, respectively. However, if the CPU parallel processing capability is sufficient, their actual execution time requires the longest execution time for any transaction. In order to determine the parallelism of two blocks, we propose the transaction parallelism rate.

$$f'(n,m) = \begin{cases} (n-m)/m & (m > 1) \\ 1 & (m = 1) \end{cases} \tag{2}$$

When $m = 1$, all the transactions can parallel execute. If all the transactions in the block are serial ($m = n$), then the transaction parallelism rate is 0; that is, no transaction can parallel execute.

4.3. Conflict Model

During the actual execution of a transaction, regarding conflicts between transactions and the number of parallel threads, there is usually the following relationship between them, as shown in Figure 7:

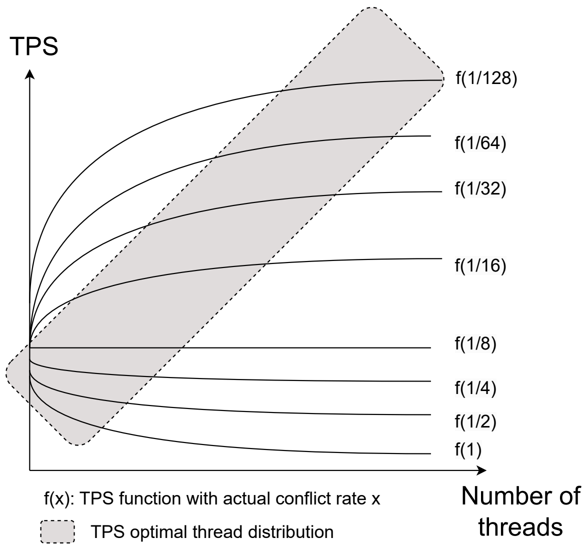


Figure 7. Estimated relationship between transaction conflicts and number of threads.

The process is shown in Appendix A. Regarding the above relationship, in summary, there are two main points:

- Under the low transaction conflict model, the number of threads is proportional to TPS; that is, in an environment with low transaction conflicts, the higher the number of threads within a certain range, the better.
- Under the high transaction conflict model, the number of threads is inversely proportional to TPS; that is, in an environment with high transaction conflicts, the lower the number of threads within a certain range, the better.

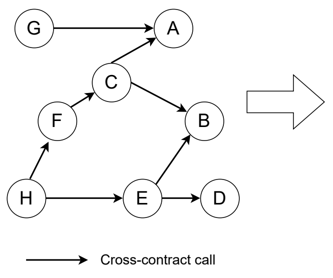
The gray area in Figure 7 is the distribution of the number of threads when TPS is optimal. In practice, this range can be obtained through multiple tests. After obtaining this range, we can use this range as the rule during an actual transaction execution to determine the number of threads based on conflict situations.

4.4. Smart Contract Dependency Build

Our scheme builds the smart contract dependency from the relationship between cross-contract calls. Through careful analysis and demonstration, we propose a set of conflict contracts that consist of four parts:

- The contract itself;
- The contract's descendant contracts (child contracts and grandchild contracts, etc.);
- The contract's parent contracts (parent contracts, grandfather contracts, etc.);
- Other parent contracts of the contract's descendant contracts.

As you can see in Figure 8, this is an example of smart contract dependency. Taking contract C as an example, contract C internally invokes contract A and contract B, but since other contracts also have corresponding operations, the actual conflicting contract set of contract C is {A, G, B, E, F, H}. That means that only contract D does not conflict with contract C.



contract	Call contract	Conflicting contract set	Parallel contract set
A	-	C、F、H、G	B、D、E
B	-	C、F、H、E	A、D、G
C	A、B	A、G、B、E、F、H	D
D	-	E、H	A、B、C、F、G
E	B、D	D、B、C、F、H	A、G
F	C	C、A、G、B、E、H	D
G	A	A、C、F、H	B、D、E
H	E、F	E、D、B、F、C、A、G	-

Figure 8. Example of smart contract dependency.

The blockchain system needs to load all contracts on the current chain when the chain starts and initialize the dependencies of all contracts in the memory of the scheduling module. The details are shown as follows:

First, the blockchain system starts and initializes the scheduling module and contract dependency DAG. Then, the system will obtain the list of smart contracts from the blockchain.

Second, traverse smart contracts and use the graphs.AddNode function to build. The process of the AddNode algorithm is shown in Algorithm 1.

Algorithm 1 AddNode

```
1:  ContractNode[] = graph.Contract.DependentContracts
2:  Node = {Children: children, Parents: nil, ConfilctNodes: children}
3:  if isZero(len(graph.Roots)) then
4:      graph.Roots.Init(node)
5:  else
6:      for child in node.Children do
7:          it.ConflictNodes.add(child.ConflictNodes)
8:          child.Parents.add(it)
9:          if Roots.Exit(child) then
10:             Roots.remove(child)
11:         end if
12:     end for
13:     for conflictNode in node.ConflictNodes do
14:         conflictNode.ConflictNodes.add(node)
15:     end for
16:     Roots.add(node)
17:     Contracts.update()
18:     Conflicts.cache.update()
19: end if
```

As depicted in Algorithm 1, our proposed method manages the addition of nodes to adaptively adjust the dependency graph and handle conflicts among contracts. Initially, we initialize the contract nodes from the graph's dependent contracts (Line 1). Subsequently, a new node is created with its children defined, no parents initially and its conflict nodes set to its children (Line 2). If the graph lacks roots, the newly created node is initialized as the root (Lines 3–4). We then process each child of the new node by adding the child's conflict nodes to the new node's conflict nodes, adding the new node to the child's parents, and removing the child from the roots if it exists there (Lines 6–12). Following this, the conflict nodes of the new node have the new node added to the conflict nodes (Lines 13–15). Finally, the new node is added to the roots (Line 16), and the contracts and conflict cache of the graph are updated (Line 17).

According to this smart contract dependency, we can build the transaction DAG, which is important for the parallel scheduling of smart contracts. We will introduce the steps for building the transaction DAG in the following subsection.

4.5. Transaction DAG Building

The generation of the transaction DAG model in the block depends on the corresponding contract relationships between transactions. There are two key points to building a transaction DAG model:

1. The transaction DAG construction order depends on the order in which transactions are packaged in the block. The earlier the order, the more likely it is to be executed first.
2. When each transaction is added to the transaction DAG, it is judged whether it has a dependency relationship with all previous transactions. If it exists, it is added to the latest dependent transaction set.

Then we can build the transaction DAG as follows:

First, initialize the transaction DAG. Then traverse the list of transactions and build the map between the name of the contract and the set of txid. From this map, you can obtain the set of txid that invoke the contract. According to the map of txid and contract dependency, we can find all the conflict transactions of one transaction immediately. Then we can build the list of preorder conflict transactions, that is, the list of transaction IDs that conflict with it for transaction txId1, among all transactions smaller than it.

While traversing the list of transactions, we can select the transactions of the contracts that have no conflicts with any contracts and set these transactions as separate root nodes.

When building the transaction DAG, we can simplify the transaction DAG. For example, transactions A, B, and C have conflicts $A \to B$, and C is behind and conflicts with A and B. In this situation, we can only set C behind B and do not need to set C behind A in the transaction DAG. This process can simplify the transaction DAG and limit the cost of time.

4.6. Transaction Parallel Execution

After building the smart contract dependency and transaction DAG, it is time to execute the transaction in parallel. Before this, we need to set transactions into the channel that is used to distribute transactions. The processes responsible for the transaction execution will receive and process these transactions.

In order to parallel execute the transactions, the node builds and starts a process pool. Every separate DAG will be executed in one process. This method makes the transactions that have no conflict parallel executed in different processes. The conflict transactions will be executed in the same process.

4.7. Transaction Execution Processing

The transaction execution processing algorithm can check the executed transactions and process the conflicts. If the transaction has a conflict with the read–write set, we should update the smart contract dependency of this transaction and update the transaction DAG. After these steps, we need to re-execute the conflict transactions until all the transactions are successfully executed.

Then we develop the transaction execution processing algorithm. As shown in Algorithm 2, our proposed scheme can process the results of transactions to adaptively adjust the transaction DAG and the execution of transactions. First, we initialize the executed transactions and set the field of transactions as true (Line 2). Then we traverse all the completed transactions in reverse order (Line 4). If the transaction conflicts with the read–write set, we update the transaction DAG and mark the field of the transaction as false (Lines 5–7). After judging the conflict, we update the contract dependency window (Line 9).

Algorithm 2 ExectionProcessing

```
1:  for transaction.isExecuted() do
2:      transaction.isDone = True
3:  end for
4:  for transaction in DoneTxs do
5:      if RWSet.isConflict(transaction) then
6:          TransactionDAG.update(transaction)
7:          transaction.isDone = False
8:      end if
9:      ConflictTable.RelationMap[contract1][contract2] = isConflict
10:     ConflictFactorWindow.update(contract1,contract2)
11:     if globalSlidingWindows.needReCul() then
12:         globalSlidingWindows.flag = True
13:     end if
14: end for
15: if DoneTxs.isEmpty() then
16:     TransactionDAG.prune(transaction)
17:     processPool.add(root)
18:     DoneTxs.add(transaction)
19:     if block.judgeTxs(DoneTxs.length()) and midDAG.isEmpty() then
20:         BlockDone()
21:     end if
22: else
23:     transaction.reSend()
24: end if
25: Algorithm End
```

When updating the smart contract dependency, we need to use the sliding window to record the execution results of transactions that invoke the smart contract. In this step, we add one bit for the sliding window to record. For transactions that are detected as conflicts, we write ‘1’ in this bit; otherwise, we write ‘0’. Then we calculate the conflict factor of these two contracts (Line 10). This step is the key point of adaptation in the algorithm.

Then we judge whether the global sliding window meets the conditions for calculating the conflict rate; if so, we should mark the global sliding window as to be recalculated (Lines 11–13).

If all the executed transactions are processed, we prune the transaction DAG and the node (Line 16) and add the new root to the process pool that is used to execute transactions (Line 17). If the length of the set of ‘DoneTxs’ meets the condition and there are no transactions to be distributed, we package the block (Line 19–21). If there are transactions that are not processed, we redistribute these transaction nodes (Line 23). All the algorithm is ended.

5. Evaluation

In this section, we introduce the experiment of our adaptive parallel scheduling scheme. Based on the comparative experiment, we evaluate the performance of our scheme by some evaluation indicators, such as latency and throughput. By analyzing the experimental data, we demonstrate that our protocol meets expectations and solves the research questions we proposed in Section 1.

5.1. Experiment Setup

We introduce the basic information about our experiment setup.

Baseline. We compared our scheme with a fully parallel scheduling scheme that can be recognized as the traditional scheme. In the fully parallel scheduling scheme, the smart contracts are executed in parallel, and the contracts that have conflicts will be re-executed again.

TextBed. We deployed two scheduling schemes in Tencent Cloud environments for our experiments. The compute service instance from Tencent Cloud is equipped with an AMD EPYC 7K62 48-Core Processor CPU, 16 GB of RAM, and CentOS.

Implements. We implemented our proposed adaptive parallel scheduling scheme and conducted a comparative analysis of the latency and throughput. ChainMaker is an open-source library written in Golang that implements an assemblable architecture for permissioned blockchain [21]. We forked the code of ChainMaker and rewrote the smart contract scheduling module based on the logic of our scheduling scheme.

Testflow. We built a TestChain with a single node and solo consensus algorithm. Solo is a test mode of ChainMaker that is used to conduct full-process testing except network and consensus modules for developers. We conducted six sets of experiments at varying conflict rates. Each set of experiments was designed with transaction conflict rates set at 0%, 20%, 40%, 60%, 80%, and 100%. In each experimental set, we designated the fully parallel setting as the control group and our proposed scheme as the experimental group. Experiments were conducted separately for both the control and experimental groups, with latency and throughput being recorded for each.

5.2. Evaluation Results

In this experiment, we randomly generate a batch of transactions, and each set of transactions has a different degree of conflict. Experiments under different transaction conflict rates can test the adaptability and effectiveness of scheduling solutions in various situations. We use average latency and throughput that are always used as evaluation indicators to describe the adaptability and effectiveness of a scheduling scheme. In order to obtain more accurate experimental data, we set tens of thousands of transactions in experiments with different transaction conflict rates and recorded valid data to calculate average experimental results. For average latency, as shown in Figure 9, our scheduling scheme maintains a relatively stable and fast latency level (around 100 ms). Especially, in a high transaction conflict rate, our scheme has more than 20 times improvement in latency compared with the fully parallel scheme. The reason for this result is that, in the case of a high transaction conflict rate, conflicting transactions in the fully parallel scheme need to be executed multiple times to ensure the atomicity and non-tamperability of the blockchain system. Multiple executions lead to higher latency. Our solution can adaptively adjust the transaction scheduling strategy according to the degree of transaction conflict to produce a more efficient execution plan.

For throughput, as shown in Figure 10, our solution also has stable and high-level throughput under different transaction conflict rates. As the transaction conflict rate increases in the experiment, the throughput of the fully parallel scheme decreases significantly, almost exponentially. Based on the advantages of adaptively adjusting the scheduling strategy, our scheme can maintain stable and efficient throughput (around 860 TPS) even when the conflict rate increases.

Final Results. By conducting experiments under the transaction conflict rate, we can intuitively observe that our solution has greater advantages in both latency and throughput. From the perspective of latency data, in the case of high transaction conflict rate, our solution is one-twentieth of the average time consumption of the fully parallel scheme. From the perspective of throughput data, our scheme is 30 times better than the fully parallel scheme under the condition of high transaction conflict rate. These data all show that our solution has more advantages in transaction scheduling based on the following characteristics:

- Smart contract dependency DAG from static analysis;
- Conflict model from numerous experiments;
- Dynamically parallel scheduling;
- Technical optimization in actual implementation.

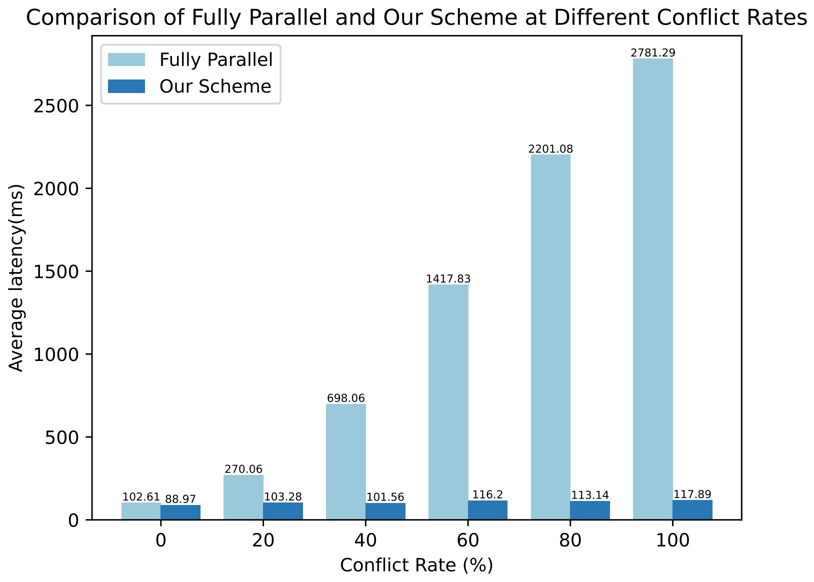


Figure 9. The latency comparison between fully parallel and our scheme.

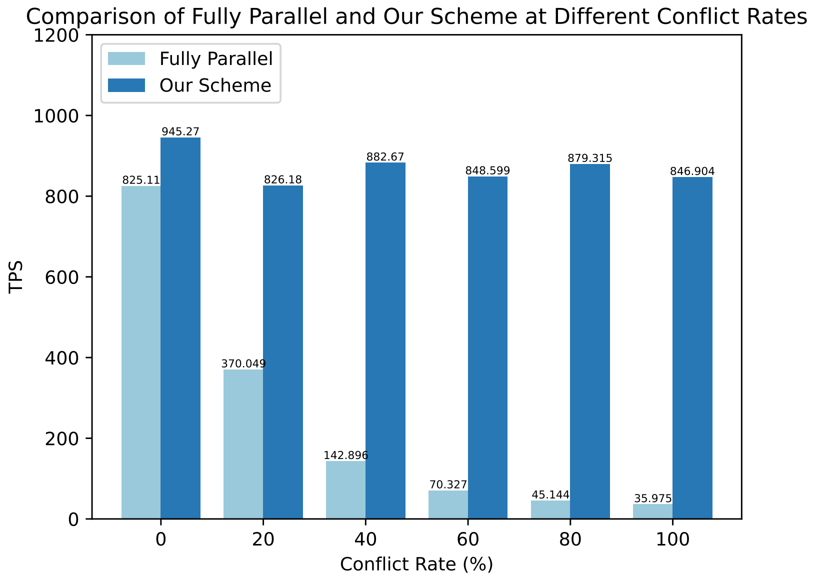


Figure 10. The throughput comparison between fully parallel and our scheme.

6. Conclusions

We propose an adaptive parallel scheduling scheme for a smart contract. By analyzing the features of a smart contract and the setting of a smart contract on ChainMaker, we build the smart contract dependency that can be used as the execution strategy. Then we propose a conflict model from the actual execution of transactions. Based on the conflict factor we define, we build the transaction DAG and execute the transactions. After the execution, we adaptively update the conflict factor and the transaction DAG to make a better execution strategy. The implementation on ChainMaker shows that our adaptive parallel scheduling scheme achieves 30 times the throughput of the traditional fully parallel scheme and one-twentieth of the average latency. Apart from the efficiency our scheme can achieve, the following are the two additional scientifically significant features: (i) Adaptability: our scheme merges static analysis and dynamic scheduling to achieve adaptability. (ii) Applicability: our scheme can be migrated to other DAG-based blockchain platforms by adjusting the data structure and implementation interface.

In the future, we will focus on adding artificial intelligence methods to the field of smart contract transaction scheduling, and use AI to train more adaptive scheduling strategies. What is more, we will try to add an intermediate layer in the architecture to achieve cross-platform compatibility.

Author Contributions: Methodology, W.Y.; software, Z.S.; writing—original draft, W.Y.; writing—review & editing, J.S., C.L. and Z.S.; visualization, G.W. and Y.L.; funding acquisition, M.A. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by the National Key Research and Development Program of China under Grant No. 2021YFB2701203, National Natural Science Foundation of China (NSFC) under Grant No. 62172040, and Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing.

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors declare no conflicts of interest. Authors Meng Ao and Zhuguang Shao were employed by the company Tencent Inc. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest. The Tencent Inc. had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. Experiments of Conflict Model

After a lot of research, we found that before a block is generated to generate a DAG, the scheduling module will execute all transactions in the block in advance and generate a transaction read–write set. If transactions are executed serially, there will be no conflicts between transactions. However, conflicts will occur if executed in parallel. For example, the same account transfers funds to two other accounts in two transactions. If executed in parallel, the two transactions will read the same account balance. Assuming that the transaction conflict rate of a certain block is particularly high, then since most transactions conflict, the execution efficiency of the scheduling module will be very low. However, assuming that the transaction conflict rate of a certain block is particularly low, most transactions will not conflicts; then a moderate increase in thread pool capacity will significantly improve overall performance. We designed a set of experiments and proposed the conflict model used in this article based on the experimental results.

Then we propose a dynamic thread pool adjustment method. Initially, the block transaction pool and the dynamic thread pool are initialized, with an initial capacity set for the latter. To regulate the thread pool capacity dynamically, a DescendCoefficient is defined. Subsequently, a sliding window bitmap is created to continually update and maintain the conflict rate throughout the adjustment process. Transactions are continuously retrieved from the block transaction pool for execution. After each transaction execution, the sliding window bitmap is updated to capture the recent conflict rate. The system monitors whether all bits in the sliding window have been updated, indicating the completion of a window's worth of transaction count since the last conflict rate calculation. If so, the latest conflict rate is calculated. Based on this rate, adjustments to the thread pool capacity are made: if the conflict rate increases, the pool capacity is decreased proportionally using DescendCoefficient; conversely, if it decreases, the capacity is increased. Prior to any modifications, upper and lower bounds are enforced to ensure that the new pool capacity remains within permissible limits. Finally, the thread pool capacity is updated, allowing for an optimized resource allocation to effectively manage transaction processing in the blockchain network. This adaptive approach ensures that the system remains responsive to varying transaction loads, thereby enhancing overall system performance and scalability. The algorithm process diagram is shown in Figure A1.

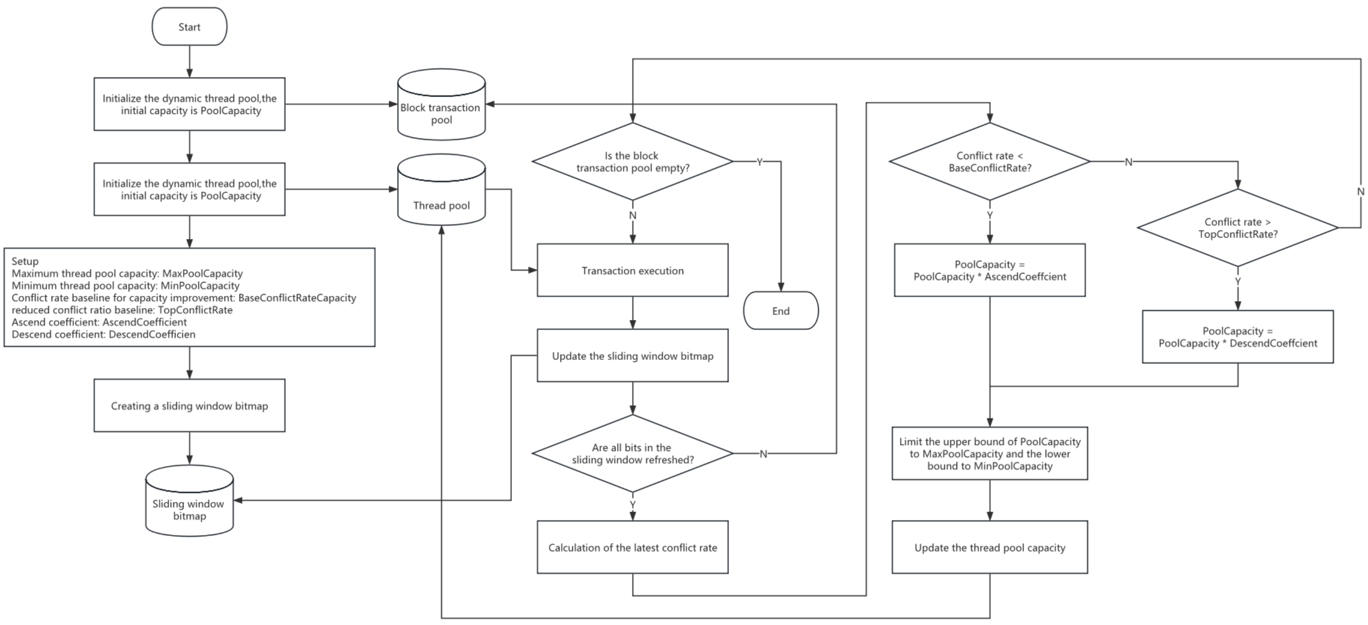
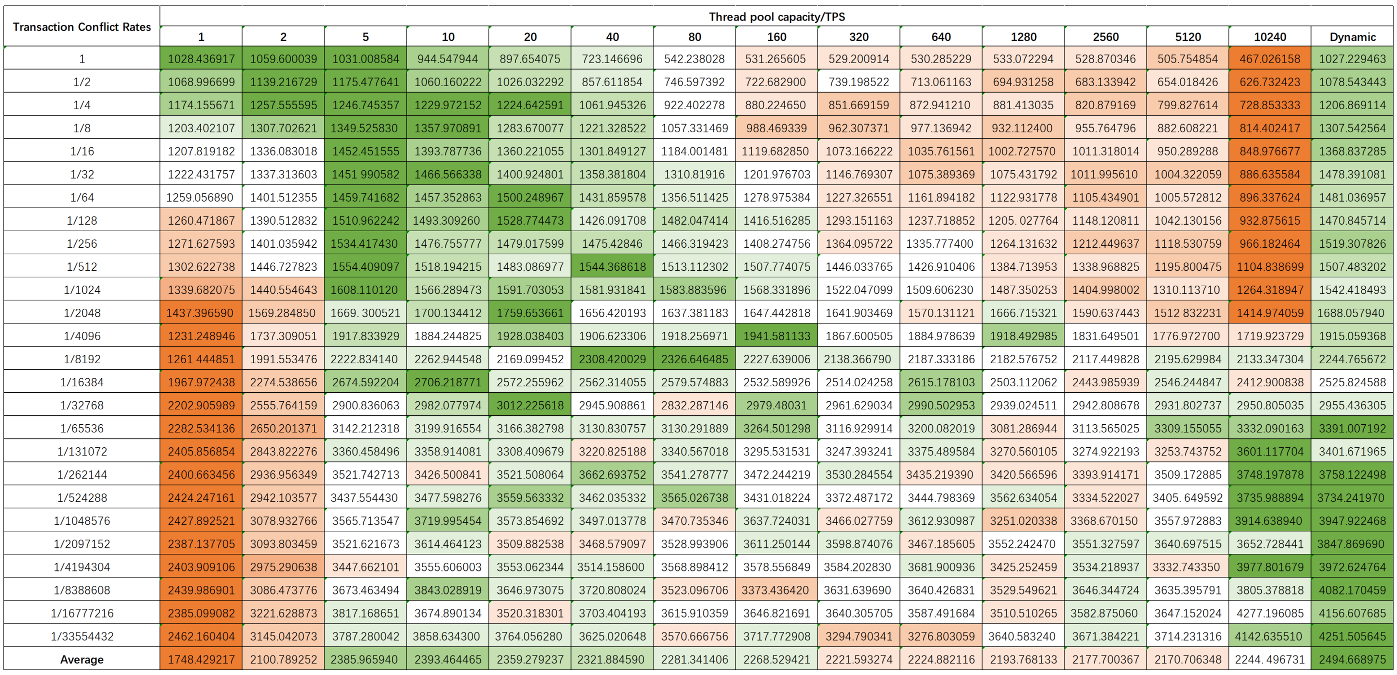

Start
Initialize the dynamic thread pool,the initial capacity is PoolCapacity
Initialize the dynamic thread pool,the initial capacity is PoolCapacity
Setup
Maximum thread pool capacity: MaxPoolCapacity
Minimum thread pool capacity: MinPoolCapacity
Conflict rate baseline for capacity improvement: BaseConflictRateCapacity
reduced conflict ratio baseline: TopConflictRate
Ascend coefficient: AscendCoefficient
Descend coefficient: DescendCoefficien
Creating a sliding window bitmap
Sliding window bitmap
Block transaction pool
Thread pool
Is the block transaction pool empty?
Y
N
Transaction execution
End
Update the sliding window bitmap
Are all bits in the sliding window refreshed?
N
Y
Calculation of the latest conflict rate
Conflict rate < BaseConflictRate?
N
Y
PoolCapacity = PoolCapacity * AscendCoeffcient
Conflict rate > TopConflictRate?
N
Y
PoolCapacity = PoolCapacity * DescendCoeffcient
Limit the upper bound of PoolCapacity to MaxPoolCapacity and the lower bound to MinPoolCapacity
Update the thread pool capacity

Figure A1. The algorithm process of dynamic thread pool adjustment.

To evaluate our dynamic thread pool adjustment method, we built an experiment on a computer service instance from Tencent Cloud. The instance was equipped with Linux, Intel Xeon Fold 6230 CPU@2.10 GHz, and 256 GB memory. The experimental setup for the conducted test is as follows: Client configuration parameters were established, including a concurrent thread count of 50,000, with each thread making a single call to the designated contract 'evm–balance' using the 'Invoke' execution method, resulting in an increment of asset value of 1. The blockchain environment was configured with Solo consensus on a single node, utilizing 80 CPU cores. Each block was designed to accommodate up to 10,000 transactions, with a maximum block size of 20 MB and a maximum block interval set at 10 s.

Furthermore, a dynamic adjustment scheme was implemented to regulate the thread pool capacity based on observed conflict rates. Specifically, when the conflict rate fell below 0.05, the thread pool capacity was increased by a factor of 3, capped at a maximum value equivalent to the block size of 10,000 transactions. Conversely, if the conflict rate surpassed 0.2, the thread pool capacity was reduced by one-fourth, with the minimum threshold set at 2 threads.

The experiment data are shown in Figure A2. The table depicts a comprehensive analysis of test parameters with varying conflict transaction ratios. The vertical axis represents the proportion of conflicting transactions, denoted, for instance, as 1/4, indicating that within a batch of sent transactions, only one out of four transactions encounters conflicts. This scenario is akin to randomly generating Invoke transactions from four different accounts, where the transactions' read–write sets do not overlap. On the horizontal axis, various test parameters are enumerated. The final column illustrates the dynamic thread pool scenario, while the preceding columns represent scenarios with fixed thread pool sizes. Within the graphical content area, each row corresponds to a different test parameter configuration. Shades of green indicate favorable outcomes, whereas shades of red signify less desirable results. The last row aggregates data for different conflict transaction ratios, providing the average transactions per second (TPS) across the tested configurations.



Transaction Conflict Rates	Thread pool capacity/TPS														
	1	2	5	10	20	40	80	160	320	640	1280	2560	5120	10240	Dynamic
1	1028.436917	1059.600039	1031.008584	944.547944	897.654075	723.146696	542.238028	531.265605	529.200914	530.285229	533.072294	528.870346	505.754854	467.026158	1027.229463
1/2	1068.996699	1139.216729	1175.477641	1060.160222	1026.032292	857.611854	746.597392	722.682900	739.198522	713.061163	694.931258	683.133942	654.018426	626.732423	1078.543443
1/4	1174.155671	1257.555595	1246.745357	1229.972152	1224.642591	1061.945326	922.402278	880.224650	851.669159	872.941210	881.413035	820.879169	799.827614	728.853333	1206.869114
1/8	1203.402107	1307.702621	1349.525830	1357.970891	1283.670077	1221.328522	1057.331469	988.469339	962.307371	977.136942	932.112400	955.764796	882.608221	814.402417	1307.542564
1/16	1207.819182	1336.083018	1452.451555	1393.787736	1360.221055	1301.849127	1184.001481	1119.682850	1073.166222	1035.761561	1002.727570	1011.318014	950.289288	848.976677	1368.837285
1/32	1222.431757	1337.313603	1451 990582	1466.566338	1400.924801	1358.381804	1310.81916	1201.976703	1146.769307	1075.389369	1075.431792	1011.995610	1004.322059	886.635584	1478.391081
1/64	1259.056890	1401.512355	1459.741682	1457.352863	1500.248967	1431.859578	1356.511425	1278.975384	1227.326551	1161.894182	1122.931778	1105.434901	1005.572812	896.337624	1481.036957
1/128	1260.471867	1390.512832	1510.962242	1493.309260	1528.774473	1426.091708	1482.047414	1416.516285	1293.151163	1237.718852	1205. 027764	1148.120811	1042.130156	932.875615	1470.845714
1/256	1271.627593	1401.035942	1534.417430	1476.755777	1479.017599	1475.42846	1466.319423	1408.274756	1364.095722	1335.777400	1264.131632	1212.449637	1118.530759	966.182464	1519.307826
1/512	1302.622738	1446.727823	1554.409097	1518.194215	1483.086977	1544.368618	1513.112302	1507.774075	1446.033765	1426.910406	1384.713953	1338.968825	1195.800475	1104.838699	1507.483202
1/1024	1339.682075	1440.554643	1608.110120	1566.289473	1591.703053	1581.931841	1583.883596	1568.331896	1522.047099	1509.606230	1487.350253	1404.998002	1310.113710	1264.318947	1542.418493
1/2048	1437.396590	1569.284850	1669. 300521	1700.134412	1759.653661	1656.420193	1637.381183	1647.442818	1641.903469	1570.131121	1666.715321	1590.637443	1512 832231	1414.974059	1688.057940
1/4096	1231.248946	1737.309051	1917.833929	1884.244825	1928.038403	1906.623306	1918.256971	1941.581133	1867.600505	1884.978639	1918.492985	1831.649501	1776.972700	1719.923729	1915.059368
1/8192	1261 444851	1991.553476	2222.834140	2262.944548	2169.099452	2308.420029	2326.646485	2227.639006	2138.366790	2187.333186	2182.576752	2117.449828	2195.629984	2133.347304	2244.765672
1/16384	1967.972438	2274.538656	2674.592204	2706.218771	2572.255962	2562.314055	2579.574883	2532.589926	2514.024258	2615.178103	2503.112062	2443.985939	2546.244847	2412.900838	2525.824588
1/32768	2202.905989	2555.764159	2900.836063	2982.077974	3012.225618	2945.908861	2832.287146	2979.48031	2961.629034	2990.502953	2939.024511	2942.808678	2931.802737	2950.805035	2955.436305
1/65536	2282.534136	2650.201371	3142.212318	3199.916554	3166.382798	3130.830757	3130.291889	3264.501298	3116.929914	3200.082019	3081.286944	3113.565025	3309.155055	3332.090163	3391.007192
1/131072	2405.856854	2843.822276	3360.458496	3358.914081	3308.409679	3220.825188	3340.567018	3295.531531	3247.393241	3375.489584	3270.560105	3274.922193	3253.743752	3601.117704	3401.671965
1/262144	2400.663456	2936.956349	3521.742713	3426.500841	3521.508064	3662.693752	3541.278777	3472.244219	3530.284554	3435.219390	3420.566596	3393.914171	3509.172885	3748.197878	3758.122498
1/524288	2424.247161	2942.103577	3437.554430	3477.598276	3559.563332	3462.035332	3565.026738	3431.018224	3372.487172	3444.798369	3562.634054	3334.522027	3405. 649592	3735.988894	3734.241970
1/1048576	2427.892521	3078.932766	3565.713547	3719.995454	3573.854692	3497.013778	3470.735346	3637.724031	3466.027759	3612.930987	3251.020338	3368.670150	3557.972883	3914.638940	3947.922468
1/2097152	2387.137705	3093.803459	3521.621673	3614.464123	3509.882538	3468.579097	3528.993906	3611.250144	3598.874076	3467.185605	3552.242470	3551.327597	3640.697515	3652.728441	3847.869690
1/4194304	2403.909106	2975.290638	3447.662101	3555.606003	3553.062344	3514.158600	3568.898412	3578.556849	3584.202830	3681.900936	3425.252459	3534.218937	3332.743350	3977.801679	3972.624764
1/8388608	2439.986901	3086.473776	3673.463494	3843.028919	3646.973075	3720.808024	3523.096706	3373.436420	3631.639690	3640.426831	3529.549621	3646.344724	3635.395791	3805.378818	4082.170459
1/16777216	2385.099082	3221.628873	3817.168651	3674.890134	3520.318301	3703.404193	3615.910359	3646.821691	3640.305705	3587.491684	3510.510265	3582.875060	3647.152024	4277.196085	4156.607685
1/33554432	2462.160404	3145.042073	3787.280042	3858.634300	3764.056280	3625.020648	3570.666756	3717.772908	3294.790341	3276.803059	3640.583240	3671.384221	3714.231316	4142.635510	4251.505645
Average	1748.429217	2100.789252	2385.965940	2393.464465	2359.279237	2321.884590	2281.341406	2268.529421	2221.593274	2224.882116	2193.768133	2177.700367	2170.706348	2244. 496731	2494.668975

Figure A2. The experiment result of the conflict model. (Shades of green indicate favorable outcomes, whereas shades of red signify less desirable results).

From the test results, it is evident that the dynamic thread pool demonstrates excellent performance across all conflict scenarios. Specifically, in situations of complete conflict, the efficiency of the dynamic thread pool surpasses that of a fixed thread pool with a size of 10240 by a factor of 2.2. Conversely, in scenarios with minimal conflict, the dynamic thread pool outperforms a single thread setup by a factor of 1.43. These data underscore the effectiveness of the dynamic thread pool in adapting to varying levels of transaction conflicts, thereby enhancing overall system efficiency and throughput in the blockchain environment.

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. Decentralized Business Review. 2008. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 8 April 2024).
2. Szabo, N. Formalizing and securing relationships on public networks. *First Monday* **1997**, *2*, e548. [CrossRef]
3. Alrubei, S.M.; Ball, E.A.; Rigelsford, J.M.; Willis, C.A. Latency and performance analyses of real-world wireless IoT-blockchain application. *IEEE Sens. J.* **2020**, *20*, 7372–7383. [CrossRef]
4. Wang, Q.; Li, R.; Wang, Q.; Chen, S. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. *arXiv* **2021**, arXiv:2105.07447.
5. Mukhopadhyay, U.; Skjellum, A.; Hambolu, O.; Oakley, J.; Yu, L.; Brooks, R. A brief survey of cryptocurrency systems. In Proceedings of the 2016 14th Annual Conference on Privacy, Security and Trust (PST), Auckland, New Zealand, 12–14 December 2016; pp. 745–752.
6. Guo, B.; Lu, Z.; Tang, Q.; Xu, J.; Zhang, Z. Dumbo: Faster asynchronous bft protocols. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 9–13 November 2020; pp. 803–818.
7. Zhang, Z.; Liu, X.; Feng, K.; Wan, M.; Li, M.; Dong, J.; Zhu, L. Phantasm: Adaptive Scalable Mining Toward Stable BlockDAG. *IEEE Trans. Serv. Comput.* **2023**, *early access*. [CrossRef]
8. Zhang, Z.; Liu, X.; Li, M.; Yin, H.; Zhu, L.; Khoussainov, B.; Gai, K. HCA: Hashchain-based Consensus Acceleration via Re-voting. *IEEE Trans. Dependable Secur. Comput.* **2023**, *21*, 775–788. [CrossRef]
9. Zhang, Z.; Feng, K.; Chen, X.; Liu, X.; Sun, H. RHCA: Robust HCA via Consistent Revoting. *Mathematics* **2024**, *12*, 593. [CrossRef]
10. Dickerson, T.; Gazzillo, P.; Herlihy, M.; Koskinen, E. Adding concurrency to smart contracts. In Proceedings of the ACM Symposium on Principles of Distributed Computing, Washington, DC, USA, 25–27 July 2017; pp. 303–312.
11. Anjana, P.S.; Kumari, S.; Peri, S.; Rathor, S.; Somani, A. An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. *arXiv* **2018**, arXiv:1809.01326.
12. Yu, W.; Luo, K.; Ding, Y.; You, G.; Hu, K. A parallel smart contract model. In Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence, Vienna, Austria, 25–26 July 2018; pp. 72–77.

13. Bartoletti, M.; Galletta, L.; Murgia, M. A true concurrent model of smart contracts executions. In Proceedings of the International Conference on Coordination Languages and Models, Valletta, Malta, 15–19 June 2020; pp. 243–260.
14. Li, H.; Chen, Y.; Shi, X.; Bai, X.; Mo, N.; Li, W.; Guo, R.; Wang, Z.; Sun, Y. FISCO-BCOS: An Enterprise-grade Permissioned Blockchain System with High-performance. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12–17 November 2023; pp. 1–17.
15. Chainmaker. Available online: https://chainmaker.org.cn/home (accessed on 8 April 2024).
16. Liu, J.; Li, P.; Cheng, R.; Asokan, N.; Song, D. Parallel and asynchronous smart contract execution. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *33*, 1097–1108. [CrossRef]
17. Digitale, J.C.; Martin, J.N.; Glymour, M.M. Tutorial on directed acyclic graphs. *J. Clin. Epidemiol.* **2022**, *142*, 264–267. [CrossRef] [PubMed]
18. Jin, C.; Pang, S.; Qi, X.; Zhang, Z.; Zhou, A. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Trans. Knowl. Data Eng.* **2021**, *34*, 5070–5083. [CrossRef]
19. Piduguralla, M.; Chakraborty, S.; Anjana, P.S.; Peri, S. An Efficient Framework for Execution of Smart Contracts in Hyperledger Sawtooth. *arXiv* **2023**, arXiv:2302.08452.
20. Miller, A. Permissioned and permissionless blockchains. In *Blockchain for Distributed Systems Security*; Wiley-IEEE Press: Hoboken, NJ, USA, 2019; pp. 193–204.
21. ChainmakerCode Homepage. Available online: https://git.chainmaker.org.cn/ (accessed on 8 April 2024).