High-Performance Smart Contracts Concurrent Execution for Permissioned Blockchain Using SGX

Min Fang[1], Zhao Zhang[1,2], Cheqing Jin[1]✉, and Aoying Zhou[1]
[1]School of Data Science and Engineering, East China Normal University
[2]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology
mfang@stu.ecnu.edu.cn, {zhzhang,cqjin,ayzhou}@dase.ecnu.edu.cn

2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00175

***Abstract*—Since there are no security concerns such as Sybil attacks, selfish mining, etc., the higher the system throughput, the better for the permissioned blockchain. And with the emergence of consensus algorithms, the throughput rates of permissioned blockchain can be up to thousands of transactions per second. The existing serial execution method for smart contracts becomes a new bottleneck for the system. Due to the lack of mutual trust between nodes, for a batch of smart contracts contained in a block, the traditional two-phase smart contract concurrency approach can only achieve concurrency within a single node, but not the parallel execution of contracts between nodes. In this paper, we propose a new two-phase framework based on trusted hardware Intel SGX, which can avoid the re-execution of all smart contracts on all nodes and improve parallelism between nodes. And consistency between nodes is achieved directly with state replication, rather than by re-executing transactions. We design a pre-execution mechanism for smart contracts in untrusted memory to batch fetch all the state data that a smart contract needs to access to reduce frequent enclave transitions during smart contract execution. Besides, we propose a method that generates a compact read-write set and a data structure named Merkle Forest which can generate the compact Merkle multiproofs for the initial data in untrusted memory in parallel and can quickly verify the correctness of the data passed in the enclave. Finally, we integrate all the techniques proposed in this paper into an open-source system BFT-SMaRt to evaluate our approach in a distributed setting. Experimental results show the efficiency of the proposed methods.**

***Index Terms*—Permissioned blockchain, smart contracts, concurrency, SGX**

I. Introduction

Smart contracts written in Turing-complete languages can describe the complicated business logic of blockchain applications, while a transaction in the blockchain is a calling of smart contracts. The main factors that affect the throughput of a blockchain system are consensus protocol efficiency and smart contract execution efficiency. Unlike permissionless blockchain system with PoW (Proof of Work) family of algorithms as a consensus protocol, where security aspects such as Sybil attacks and selfish mining need to be considered, the higher the throughput rate the better for permissioned blockchain systems with PBFT (Practical Byzantine Fault Tolerance) family of algorithms as a consensus protocol. In fact, in the permissioned blockchain system with thousands of tps (transactions per second) consensus algorithm, how to effectively execute smart contracts becomes a critical bottleneck [1]. We did a set of experiments on Quorum [2] running the BFT consensus algorithm, and results showed it can reach 2000+ tps without any transaction workload. But the throughput is down to 100 tps with real-world smart contract transactions from Ethereum [3] network.

Concurrent execution is a direct way to improve smart contract execution efficiency. Unlike concurrency control protocols widely used in databases, smart contract concurrency control in blockchain requires to consider the determinism of concurrent execution on multiple copies, concurrency conflict detection for smart contracts written in Turing-complete language, and Byzantine fault tolerance in a hostile environment. Currently, most existing solutions follow a two-phase execution framework [4]–[7], where a serializable concurrency control protocol is adapted in the first phase to pre-execute all transactions in a block on the primary node, following which both the write set and concurrent scheduling logs are transmitted to other nodes (also known as replicas). In the second phase, replicas replay transactions and validate results in parallel. Finally, the requirements are met.

However, there are two weaknesses to these approaches. First, during the first phase, all replicas are idle when the primary is executing transactions. Second, all replicas in the second phase replay the same batch of transactions. Therefore, these two-phase frameworks do not achieve full parallelism of the entire node network, due to the lack of mutual trust among nodes. Fortunately, trusted hardware has evolved rapidly in recent years and can help establish mutual trust among nodes. The trusted execution environment (TEE) such as Intel Software Guard Extensions (SGX) [8] can provide an encrypted region, also known as the enclave, to enable confidentiality and integrity protection of code and data inside it even from privileged software and physical attacks.

However, concurrent smart contract execution using SGX still has the following **challenges** under the storage limitation of Enclave Page Cache (EPC), which is limited to 128MB and only about 93MB is available [9]: **i)** How to minimize the number of costly enclave transitions and the amount of transition data [10] when the code and access state data of smart contracts are beyond the scope of EPC; **ii)** How to quickly generate integrity proof for initial data transferred from untrusted memory to enclave based on the multi-core processor when the stored data is very large; **iii)** How to quickly verify the correctness of incoming data with integrity proof in the enclave under multiple threads.

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00175

Given the above, we design a new two-phase execution framework based on SGX to improve smart contracts execution efficiency. In the first phase (*Execution* phase), we use the enclave of SGX to protect execution results of smart contracts from attack, avoiding the replicas to replay all transactions to verify results, and smart contracts are dispatched to different nodes to execute simultaneously, avoiding the idle waiting of the replicas when the primary is executing contracts. In the second phase (*Follow* phase), the trusted execution results are synchronized among nodes by state replication. Meanwhile, we use compressed data set and scheduling logs for bulk data transfer between untrusted memory and enclave to reduce the number of enclave transitions and the amount of transition data. Merkle proof [11] is used to ensure the correctness of the transmitted data. To further compress proofs, we propose a data structure named Merkle Forest to provide a certificate for data in untrusted memory, which can get and verify proofs for multiple values in parallel even for a large data set.

Contributions, our main contributions are as follows:

- We propose an efficient concurrent execution approach of smart contracts based on the traditional two-phase framework for permissioned blockchain system equipped with SGX, which avoids the re-execution of smart contracts on all nodes and improves parallelism between nodes.
- We devise a pre-execution mechanism for smart contracts in the untrusted memory to batch fetch all the state data that a smart contract needs to access to minimize the expensive overhead of enclave transitions during the process of smart contract execution.
- We design and implement a prototype system integrated with the above technologies in open-source BFT-SMaRt and evaluate it by using the standard benchmark in a distributed setting. Experimental results show the efficiency of the proposed methods.

Organization. The rest of this paper is organized as follows. Section II presents the overall architecture of our system model. Section III explains our SGX-based two-phase concurrent execution scheme in detail. Section IV reviews the latest works about our study. The experimental evaluations are presented in Section V. Finally, a brief conclusion is provided in Section VI.

II. System Overview

In this study, we propose a new two-phase execution framework based on SGX, which can avoid the re-execution of smart contracts on all nodes and improve parallelism between nodes. Meanwhile, we focus on the performance problem of concurrent execution of smart contracts on SGX-equipped nodes in the first *Execution* phase.

A. System Architecture

Figure 1 illustrates our SGX-based two-phase system architecture. A batch of contracts goes through the following four steps from calling to executing. i) First, clients call smart contracts by sending transactions; ii) Second, *PrimaryNode* assigns a batch of transactions contained in the current block

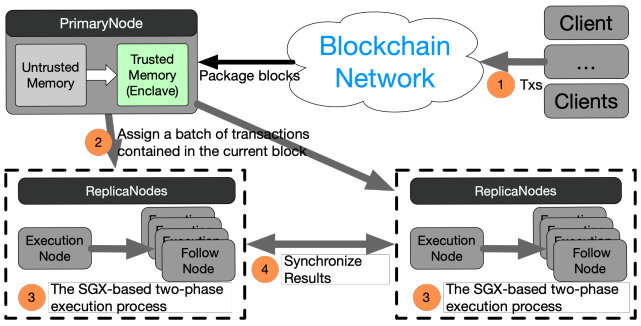


Fig. 1. System architecture

after consensus to different *ReplicaNodes* for execution according to pre-specified rules; iii) Then, following the SGX-based two-phase execution process, *ReplicaNodes* are divided into *Execution Node* and *Follow Node*, where *Execution Node* executes the transaction previously assigned to it by the *PrimaryNode* concurrently and results are synchronized directly to *Follow Node* via state replication; and iv) Finally, all nodes belonging to different task groups also synchronize data via state replication.

B. Adversary Model

In this study, all nodes are considered potential adversaries because no participant in the blockchain network trusts others. To solve this threat and make replicas get rid of the process of re-executing all smart contracts, we apply SGX to ensure the integrity of execution results. Since SGX is memory limited, all data needed in the enclave are from untrusted memory. Hence, we adopt the Merkle proof [11] to guarantee the correctness and completeness of the initial data passed into the trusted memory. We assume the probability of verification based on the wrong initial state or proof is negligible [12]. And *Follow Node* can trust the correctness and the completeness of the encrypted result because an adversary cannot break the hardware security enforcement of SGX even though it may compromise OS and other privileged software. Besides, each message prepared to pass to the enclave will be endowed with a monotonic counter to avoid a replay attack. A trusted monotonic counter also is maintained in the enclave to protect the latest version of the round. Here we use SGX monotonic counter service to guarantee the freshness of incoming data, and transactions will not be executed if the data is old.

III. EXECUTION FRAMEWORK

In the SGX-based two-phase framework, *Execution Node* pre-executes the smart contracts in untrusted memory, then all data items accessed by this batch of transactions are identified and passed into the enclave in batch, thus minimizing the number of enclave transitions. It's worth noting that we need to provide proofs for data items identified in untrustworthy memory. Figure 2 is the interaction process of untrustworthy memory and the enclave on *Execution Node*.

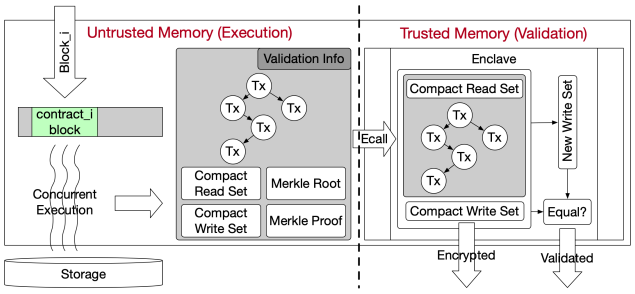


Fig. 2. The concurrent execution on an execution node

A. Pre-execution in Untrusted Memory

Pre-execution in untrusted memory has three goals, which are to generate the read-write set that the transactions need to access, to generate schedule logs for transaction execution in order to replay the same transactions in the enclave, and to generate proofs for initial data.

Generate Compact Read-Write Set. In order to improve transaction execution efficiency in the enclave, we get the read-write set at the pre-execution phase and compress it to generate a compact read-write set, i.e., the oldest read set and the latest write set for the current batch of transactions.

Generate Schedule Logs. The framework we proposed can use the concurrent smart contract execution method proposed before [4]–[7]. For a batch of transactions, we use OOC (Optimistic Concurrency Control) protocol to execute and use the graph algorithm to rollback all transactions that have read and write conflicts after current round of execution to ensure the correctness of results, just like the method used in [7].

Generate Compact Merkle Multiproofs. We adopt general Merkle proof [11] to ensure the correctness of the compact read-write set passed into the enclave. To reduce the size of proofs, we naturally think of the multi-value proof problem of the Merkle tree, which can prove multi-values exist on the same Merkle tree, namely multiple states belong to the same version at the same time. Although the existing methods about multiproof [13], [14] can save storage space to a certain extent, they still have the following three problems: i) Extra indexes to verify; ii) Difficult to generate and verify in parallel; and iii) Lack of scalability to support a large tree.

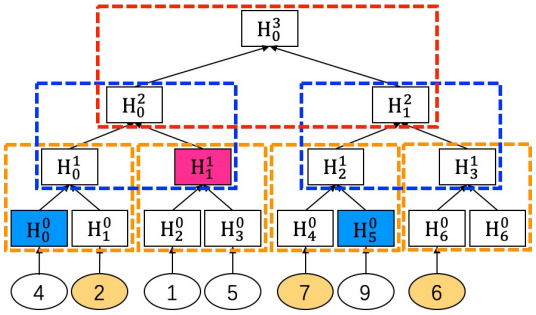


Fig. 3. The Merkle forest

To increase the parallelism of generating multiproofs for data and verifying data integrity, for a dataset D, we design a Merkle Forest consisting of multiple sub-trees with each sub-tree a specified size 2^N (N is user-defined), obviously, $D = k*2^N+m, m \in (0, 2^N)$. If the number of sub-tree exceeds 2^N we will construct a new Merkle tree above them until the root tree. Figure 3 shows a simple example of Merkle Forest with $N = 1$ and $D = 7$, which presents a hashcode with H_j^i (i is the level of tree, j is the index of $i-th$ level). At the lowest level, there are $k = 3$ full sub-tree and $m = 1$ unfull sub-tree contains value H_6^0. The roots of the lowest level are the leaves of the upper level, so we can compute the tree again until it is the root tree. Obviously, the multiproofs are completely sparse. Furthermore, for each sub-tree, we can independently calculate multiproofs and use multiproofs for verification. To obtain multiproofs on a single sub-tree as quickly as possible, we designed Algorithm 1. For the update process, we use an in-place update method, which will be completed asynchronously during the consensus process. The update process is similar to obtaining mulitiproofs, and will not be repeated here.

Algorithm 1: Generate Compact Merkle Multiproofs

Input: the Merkle tree H , and the ordered read index list of read set γ
Output: the compact Merkle multiproofs M

```
1  for each H^i in H do
2      if H^i.length = γ.length then
3          break
4      create the next level index list nli
5      for each j in γ do
6          if j is odd then
7              M^i.add(H^i_{j-1})
8          else
9              if j is not the last entry of γ then
10                 if its right neighbor in γ then
11                     skip it and its right neighbor
12                 else
13                     M^i.add(H^i_{j+1})
14             else if j is not the last index of H^i then
15                 M^i.add(H^i_{j+1})
16         nli.add(j/2)
17     γ ← nli
18 return M
```

The input of Algorithm 1 is the Merkle tree H and the ordered index set γ of the read set. Algorithm 1 iterates H from bottom to top. If the size of γ is equal to the size of H^i, it indicates that the proof of all values in $i-th$ layer needs to be calculated, i.e., the proofs in each layer after $i-th$ layer can be calculated without extra proofs, so we do not need to continue (Line 2-3). If not, we should create the next level index list of value nli and then traverse γ (Line 4-5). When

the index j of γ is odd, we need to maintain proof of its previous index (Line 7). Otherwise, we should judge whether it is the last index in the γ or not. If not, skip them when its right neighbor in γ, or add its right neighbor to M^i(Line 10-13). If so and it is not the last index of H^i, we should maintain its right neighbor in M^i (Line 15). At last, add $j/2$ to nli for the next level computation(Line 16). After iteration, we will return the compact Merkle multiproofs M.

Example 1. As shown in Figure 3, the ordered index list γ of read set is {1,4,6}. We traverse the γ in order. The first value is odd 1, so it needs the value of the node with index 0 on its left, i.e., H_0^0, to get H_0^1, and add index 0 to nli. The second value 4 is even, so it needs the value of the node with index 5 on its right, i.e., H_5^0, to get H_2^1, and add index 2 to nli. Due to the last even value 6 belongs to the last tree without a right node, it will generate H_3^1 with itself, and add index 3 to nli. Finally, we get $M^0 = \{H_0^0, H_5^0\}$ and the next $\gamma = nli = \{0, 2, 3\}$. We will continue to iterate to get $M^1 = \{H_1^1\}$. According to Algorithm 1, we will get a compact Merkle multiproof $M = \{\{H_0^0, H_5^0\}, \{H_1^1\}\}$.

B. Validation in the Enclave

In the validation phase, the compact read set would be validated by using the Merkle root and its corresponding Merkle multiproofs to predict whether this node is a Byzantine node in advance. After validated, concurrent replay these transactions based on schedule logs to get a new write set. Finally, this new write set and the incoming write set will be compared. If all is right, a certificate will be generated and passed to untrusted memory to prove that the execution result is correct and not tampered with. The encrypted output of the received write set and the whole validation process is handled asynchronously in parallel.

Validate with Compact Merkle Multiproofs. To reduce the amount of data passed into the trusted memory, we use an index-less multiproofs, and propose Algorithm 2 to accelerate the multi-value verification process. The input of Algorithm 2 is the compact Merkle multiproofs M and the read set which is divided into the index list γ and the hashcode list β. Algorithm 2 computes the proof based on M from bottom to top to get $Root$. Due to no indexes in M, we must judge whether it is the last level or not (Line 1). For each iteration, we will create the next level index list nli, and the next level value list nlv (Line 2). For each index ω in γ, if ω is odd, there must be a pair value in β, so just compute with them (Line 5). When it is even but not the tail value (Line 7), we should judge whether the next index in γ is its right neighbor or not. If so, we should compute the next value with them and skip its right neighbor(Line 9), otherwise, we should compute with a value pop from M^0 (Line 11). When ω is the tail index, we should judge whether M^0 is empty (Line 13), if so, the next value will be computed with itself (Line 16). Otherwise, it will be computed with the last value in M^0. At last, we should add the $\omega/2$ in nli and pop M^0 if M is not empty for the next level computation. After the iteration, we should assign nli to γ and nlv to β to compute the next level (Line 19). Finally, there will be only one value in β, we will return it as the $Root$.

Algorithm 2: Validate with Merkle Multiproofs

Input: the compact Merkle multiproof M, the ordered compact read index list γ and hashcode list β
Output: the computed root of M

```
1  while γ.length>1 || M is not empty() do
2      create the next level index list nli and value list nlv
3      for each ω in γ do
4          if ω is odd then
5              nlv.add(hash(M^0.pop(), β[ω]))
6          else
7              if ω is not the last entry of γ then
8                  if its right neighbor in γ then
9                      nlv.add(hash(β[ω], β[++ω]))
10                 else
11                     nlv.add(hash(β[ω], M^0.pop()))
12             else
13                 if M^0 is empty() then
14                     nlv.add(hash(β[ω], β[ω]))
15                 else
16                     nlv.add(hash(β[ω], M^0.pop()))
17         nli.add(ω/2)
18         pop M^0 if M is not empty
19     vi ← nli,vl ← nlv
20 return β[0]
```

Example 2. According to Example 1, we get $M = \{\{H_0^0, H_5^0\}, \{H_1^1\}\}$, γ ={1,4,6}. As shown in Figure 4, Based on Algorithm 2, we will traverse M in order. Firstly, We will use M^0 and γ to verify and generate the second layer proof. Because the first 1 in γ is odd, so compute H_0^1 with H_0^0 in M^0 and add index 0 to nli. The second 4 is even, so compute H_2^1 with H_5^0 and add index 2 to nli. When computing the last 6, due to no item in γ and M^0, we get H_3^1 by compute H_6 with itself and add index 3 to nli. Finally, we get the next $\gamma = nli = \{0, 2, 3\}$ and pop M^0 from M. We will continue to iterate based on this γ and M^0 until getting $Root$.

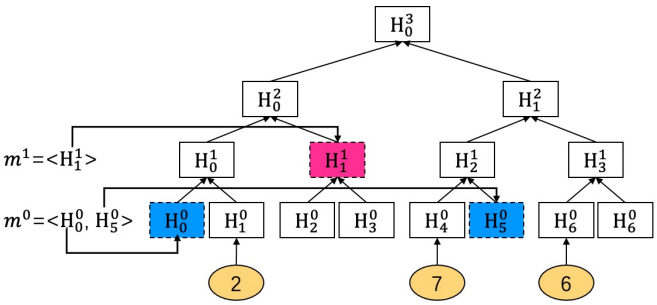


Fig. 4. Verify with multiproofs

IV. Related Work

In this section, we review the latest research works about our study, including concurrent smart contract execution, combining smart contracts with TEE, and Merkle multiproofs.

A. Concurrent Smart Contract Execution

Recently, there have been some works about the concurrent execution of smart contracts. Dickerson et al. firstly propose the concurrent execution method of smart contracts, where the miner uses two-phase lock protocol to ensure serializability of smart contract execution, and a concurrent schedule is recorded during execution and uploaded to the validators [4]. Anjana et al. replace the pessimistic lock protocol with an optimistic concurrency control protocol at the primary, independent and parallel validation is used to improve efficiency in the validation phase [6]. Zhang et al. allow miner to use any concurrency control protocol by transmitting a read-write set to validators after execution without conflict assumption [5]. Pang et al. design an efficient concurrent control mechanism for smart contracts, which optimizes both phases of contract execution [7]. All approaches mentioned above can make many nodes idle because all validators would wait the miner to compute firstly and then re-execute secondly. While the SGX-based two-phase execution framework we proposed can avoid re-execution of the same batch of transactions on all nodes and improve parallelism among nodes.

B. Smart Contracts with TEE

Several works address the security and privacy issues of designing smart contracts by using TEE. Hawk [15] provides a confidential execution environment for contracts by using cryptography and Intel SGX. Microsoft introduced an open-source blockchain framework called Coco [16]. Each node in the Coco is protected by TEE such as Intel SGX and validated before joining the network. Thus, a trusted network is formed, where each node is considered harmless. But it doesn't take into account too many engineering and implementation issues, and the performance constraints of TEE. Ekiden [17] tries to put all smart contracts calculations in TEE, which also refers to small smart contracts. None of these consider the performance of smart contract execution and TEE's performance flaws, while ours takes the overall interest of both.

C. Merkle Multiproofs

Merkle Hash Tree (MHT) [18] is widely used in blockchain [3] to verify the integrity of data. Although the way to construct an MHT is surveyed for a long time, nearly most of them care about the validation for one value. In recent years, some studies have turned to the problem of sparse Merkle multiproofs like [13], in which the proofs of all values are generated together and verified together. To further reduce the size of multiproofs, some compact Merkle multiproofs which only record the index of elements used for creating a multiproof other than an index for every non-leaf hash used before are proposed [14]. But at present, all such work cannot generate and verify multiproofs for larger data set in parallel.

V. Experiments

We implement a prototype system that integrates all the techniques proposed above with an open-source PBFT framework BFT-SMaRt [19] and evaluate its performance.

A. Experiment Setup

All experiments were conducted upon a cluster of 4 virtual processing machines, each equipped with 16 virtual cores and 16GB memory, which based on UCloud-N2. All machines running Ubuntu 18.04 with JDK version 1.8 and connected by a gigabit Ethernet switch. The project is building in debug simulation mode of Intel SGX, the enclave memory is set to the maximum 128MB, and the SGX SDK [20] is v2.6. All codes in untrusted memory and the trusted enclave are written in Java and C++, respectively, and communicate via JNI.

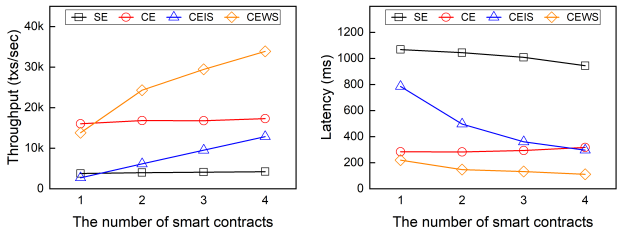
We use SmallBank [21] as the workload for testing. As a typical OLTP workload, SmallBank is widely used to evaluate blockchain systems. To simulate the real workload properly, we extend the original SmallBank benchmark by implementing two new transactions: $SendPayment$ and $Query$, where $SendPayment$ transfers money from one account to another one, and $Query$ reads the checking and the saving account of a given user. The workload is produced by uniform distribution. The database is initiated with 1000k customers. Before the start of each round of experiments, the system will be warmed up for three minutes, and all experimental figures show an average of 10 runs. Based on pre-experiment, we fix the size of a sub-tree (the number of leaf nodes involved) to be 2^{10}.

In the following experiments, we use SE, CE, $CEIS$, and $CEWS$ to respectively denote serial execution, concurrent execution, concurrent execution in SGX (put execution completely in SGX), and our concurrent execution with SGX.

B. Experimental Reports

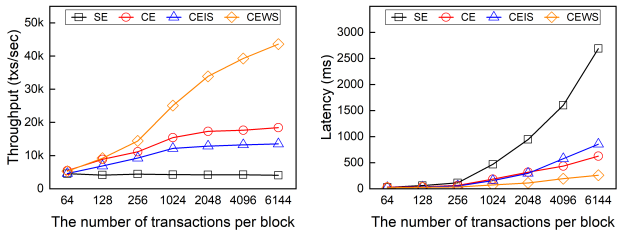
1) Varying the number of smart contracts: We evaluate the scalability of CEWS by varying the number of smart contracts from 1 to 4, where transaction count is fixed to 2,048 transactions per block with fixed 16 threads. Figure 5 reports the throughput and latency against the number of smart contracts. Under all situations, SE and CE basically have no change in throughput and latency. In contrast, the throughput of CEWS continues to rise and does not show any flattening trend. Therefore, as the number of contracts increases, CEWS can complete the calculation with a higher degree of parallelism. At the same time, SE, CE, and CEIS all have higher latency than CEWS.

2) Varying the number of transactions per block: Figure 6 reports the throughput and latency of different system upon fixed 4 smart contracts and 16 threads when the blocksize varies from 64 transactions to 6,144 transactions. As we can see, increasing the blocksize also increases the throughput of CE, CEIS and CEWS. So is the latency. This is because the fact that the usage of larger blocks causes less network communication, but leads to greater execution overhead. Obviously, CEWS gains higher throughput and lower latency over SE, CE and CEIS with an increase in the blocksize.



(a) Throughput (b) Latency

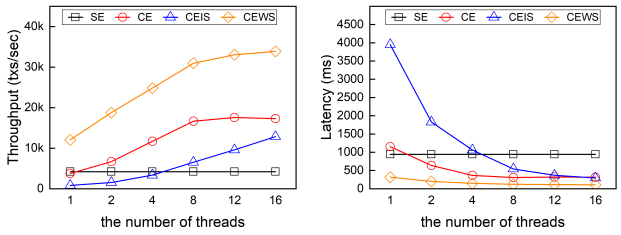
Fig. 5. Throughput and Latency against the number of smart contracts



(a) Throughput (b) Latency

Fig. 6. Throughput and Latency against the number of transactions

3) Varying the number of threads: Figure 7 shows the throughput and latency against the number of threads upon fixed 4 smart contracts, 2,048 transactions per block when the number of threads varies from 1 to 16. Obviously, the throughput of CE, CEIS and CEWS will increase as the number of threads increases. The latency of CEIS, CEWS and CE will decrease as the number of threads grows and will be much smaller than SE. This gives us an impression of the effectiveness of CEWS, which benefits from more threads.



(a) Throughput (b) Latency

Fig. 7. Throughput and Latency against the number of threads

VI. Conclusion

In this paper, we present an efficient SGX-based two-phase execution framework of smart contracts for permissioned blockchain aiming at higher parallelism and throughput. We focus on the performance problem of smart contracts concurrent execution on SGX-equipped nodes due to the limited size of enclave and the expensive cost of enclave transitions and paging. We devise a pre-execution mechanism for smart contracts in untrusted memory to batch fetch all state data that a smart contract needs to access to minimize the expensive overhead of enclave transitions during execution. Meanwhile, a data structure called Merkle Forest supported a large amount of data is proposed, which can quickly obtain compact Merkle multiproofs and verify data in parallel.

Acknowledgment

This work is partially supported by National Science Foundation of China (U1811264, U1911203, 61972152 and 61532021) and Guangxi Key Laboratory of Trusted Software (kx202005). Cheqing Jin is the corresponding author. The authors would like to thank the anonymous reviewers for their valuable feedback.

References

[1] M. Vukolić, "Rethinking permissioned blockchains," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. ACM, 2017, pp. 3–7.
[2] Quorum. https://github.com/jpmorganchase/quorum.
[3] Ethereum. https://github.com/ethereum.
[4] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2017, pp. 303–312.
[5] A. Zhang and K. Zhang, "Enabling concurrency on smart contracts using multiversion ordering," in *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 2018, pp. 425–439.
[6] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 83–92.
[7] S. Pang, X. Qi, Z. Zhang, C. Jin, and A. Zhou, "Concurrency protocol aiming at high performance of execution and replay for smart contracts," *arXiv preprint arXiv:1905.07169*, 2019.
[8] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.
[9] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A performance analysis tool for intel sgx enclaves," in *Proceedings of the 19th International Middleware Conference*. ACM, 2018, pp. 201–213.
[10] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 81–93, 2017.
[11] R. C. Merkle, "Method of providing digital signatures," Jan. 5 1982, uS Patent 4,309,569.
[12] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated join processing in outsourced databases," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009, pp. 5–18.
[13] Sparse merkle multiproofs. https://www.wealdtech.com/articles/understanding-sparse-merkle-multiproofs/.
[14] L. Ramabaja and A. Avdullahu, "Compact merkle multiproofs," *arXiv*, pp. arXiv–2002, 2020.
[15] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
[16] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna *et al.*, "Ccf: A framework for building confidential verifiable replicated services," Technical Report MSR-TR-2019-16, Microsoft, Tech. Rep., 2019.
[17] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.
[18] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.
[19] Bft-smart. https://github.com/bft-smart/library.
[20] Intel sgx sdk. https://software.intel.com/en-us/sgx/sdk.
[21] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 4, p. 20, 2009.