Operation-level Concurrent Transaction Execution for Blockchains

Haoran Lin
haoran_lin@zju.edu.cn
Zhejiang University

Yajin Zhou
yajin@blocksec.com
Zhejiang University
BlockSec

Lei Wu
lwu@blocksec.com
Zhejiang University
BlockSec

Abstract

Despite the success in various scenarios, blockchain systems, especially EVM-compatible ones that serially execute transactions, still face the significant challenge of limited throughput. Concurrent transaction execution is a promising technique to accelerate transaction processing and increase the overall throughput. Existing concurrency control algorithms, however, fail to obtain enough speedups in real-world blockchains due to the high-contention workloads.

In this paper, we propose a novel operation-level concurrency control algorithm designed for blockchains. The core idea behind our algorithm is that only operations depending on conflicts should be executed serially, while all other conflict-free operations can be executed concurrently. Therefore, in contrast to the traditional approaches, which block or abort the entire transaction when encountering conflicts, our algorithm introduces a redo phase to resolve conflicts at the operation level by re-executing conflicting operations only. We also develop a set of data dependency tracking mechanisms to achieve precise identification and speedy re-execution for conflicting operations. We implement an open-source prototype based on Go Ethereum and evaluate it using real-world Ethereum blocks. The evaluation results show that our algorithm achieves an average speedup of 4.28×. If combined with state prefetching techniques, our approach can further accelerate the transaction execution by 7.11×.

1. Introduction

Following the growing prosperity of cryptocurrencies [36, 48], blockchain technologies have gained increasing attention. The introduction of programmable smart contracts [13] has further expanded the applications of blockchains beyond cryptocurrencies, e.g., decentralized finance. But the limited throughput remains one of the most critical issues of blockchains, hindering their wider adoption.

The throughput of blockchains can be increased by either shortening block time or enlarging block size. Block time (i.e., the average time to generate a new block) is typically determined by the consensus protocol. In recent years, researchers have proposed many consensus protocols capable of generating a block in a short time [30, 31, 44, 45]. Ethereum [48], one of the most popular blockchain platforms, has already switched to the efficient Proof-of-Stake (PoS) consensus [24] from the slow Proof-of-Work (PoW) consensus [36]. With these advances in consensus protocols, the block time is no longer the throughput bottleneck. On the other hand, block size is determined by the number of transactions executed within the block time. If the transaction execution speed remains the same, a shorter block time eventually leads to a smaller block size, which is unfavorable for the overall throughput. Therefore, optimizing consensus protocols alone does not necessarily improve the throughput. But if the transaction execution bottleneck is alleviated, blockchains can be configured with both short block time and large block size, thus increasing throughput.

Concurrent Execution in Blockchains. One of the most promising techniques to accelerate the execution layer of blockchains is concurrent execution. It harnesses the parallelism of commodity hardware, which is untapped under the serial execution model, to speed up transaction execution. Recent works [2, 8, 18, 23] have attempted to adopt traditional concurrency control algorithms to blockchains. But they fail to obtain enough speedups in real-world blockchain workloads due to the hot spot problem (that causes a large number of conflicts).

Blockchain workloads feature highly skewed data accesses, making hot spots common. For example, in Ethereum workloads, 62% storage operations access only 0.1% of storage slots. Moreover, most storage operations in blockchains follow the read-modify-write (RMW) pattern. Multiple concurrent RMW operations accessing data hot spots bring about severe data contention and lead to significant performance degradation for traditional concurrency control algorithms. Specifically, pessimistic algorithms, such as two-phase locking (2PL) [10], only obtain low concurrency due to the lock contention on hot spots; while optimistic algorithms, such as optimistic concurrency control (OCC) [28], encounter frequent aborts due to data conflicts on hot spots.

The root causes behind the poor performance of these traditional algorithms are their transaction-level conflict-handling strategies. They view transactions as indivisible and treat internal operations as "black boxes." When encountering a conflict, they have to block or abort the entire transaction, regardless of how many operations are affected by the conflict. Such coarse-grained parallelizing strategies are inefficient, especially in high-contention blockchain workloads.
Key Insights. To motivate a new conflict-handling strategy, let us consider a typical token transfer example shown in Figure 1. Transaction tx1 manifests a transfer of ten tokens from user A to B, and transaction tx2 manifests a transfer of ten tokens from user A to C. These two transactions conflict because they both read, modify, and write A’s balance. But it is worth noting that the data conflict on A’s balance does not influence the balances of B and C. They are increased by ten tokens regardless of the conflict, as long as A has enough tokens to complete both tx1 and tx2. In other words, the RMW operations on the balances of B and C are conflict-free and thus can be executed concurrently without blocking or aborting. To achieve this, a possible optimistic execution schedule can be: (i) execute tx1 and tx2 concurrently and speculatively just as OCC, (ii) validate and commit tx1, (iii) detect that tx2 misses tx1’s update on A’s balance, and (iv) re-execute the RMW operations on A’s balance in tx2 and then commit tx2. In this way, we resolve data conflicts at the operation level. That is, instead of blocking or aborting the entire transaction, we only re-execute operations that depend on the conflicting data (i.e., the update on A’s balance in tx2 in this example).

Our Approach. With the insights above, we propose a novel operation-level concurrent control algorithm capable of exploiting parallelism even in high-contention blockchain workloads. The central idea behind our algorithm is to resolve data conflicts at the operation level: only operations depending on the conflicting data need to be executed serially, while all other conflict-free operations can be executed concurrently without blocking or aborting.

Specifically, we use a variant of optimistic concurrency control (OCC). Like OCC, our algorithm first executes transactions concurrently and speculatively while tracking storage slots they read and write in the thread-local memory. Then our algorithm validates and commits these transactions one by one. If a transaction’s read set is consistent with the committed storage, our algorithm commits the transaction and installs all values in its write set. Otherwise (i.e., conflicts are detected), rather than aborting the transaction as OCC, our algorithm introduces a novel redo phase to identify and re-execute all operations influenced by the conflicts. In this way, our algorithm resolves data conflicts by re-executing only several operations, which largely mitigates the performance degradation caused by data contention.

Moreover, to achieve conflicting operation identification and re-execution in the redo phase, we propose a technique called SSA (static single assignment) operation log. Unlike stack-based EVM instruction traces, our SSA operation logs not only record the inputs and outputs of the executed operations but also maintain the dependencies (i.e., use-definition and definition-use chains) among operations. With the help of the SSA operation log, we can identify all conflicting operations by traversing the definition-use chains, retrieve their inputs from the definition operations, and finally re-execute them.

We have implemented a prototype system, Olive, based on the Go Ethereum [11], and we will release the source code to engage the community1. We conduct a comprehensive evaluation of our prototype system. The evaluation results show that Olive achieves an average speedup of $4.28 \times$ on real-world Ethereum blocks. Furthermore, if combined with state prefetching techniques, the speedup can reach $7.11 \times$.

Our Contribution. In summary, this paper makes the following contributions:

- **Novel Strategy.** We introduce a novel operation-level conflict-handling strategy as the enabling mechanism for parallelizing transaction operations even in high-contention blockchain workloads.
- **New Techniques.** We propose a new SSA operation log for conflicting operation identification and re-execution and an effective data-flow tracking algorithm for the SSA operation log generation.
- **Efficient Prototype.** We implement a prototype system, Olive, which can process real-world Ethereum transactions in parallel. To the best of our knowledge, Olive is the first implementation of Ethereum Virtual Machine (EVM) that supports operation-level concurrent transaction execution.
- **Thorough Evaluation.** We evaluate Olive using real-world Ethereum blocks. The results demonstrate that Olive accelerates the execution layer of Ethereum effectively and creates a real opportunity to increase the overall throughput.

2. Background

2.1. Blockchain

A blockchain is a replicated state machine shared across computers on a peer-to-peer network. It consists of an ordered list of blocks linked via cryptographic hash pointers. Each block further comprises a sequence of transactions, which either transfer cryptocurrency or invoke smart contract [48] code to modify the blockchain state. The design of a blockchain system typically involves two independent layers: the consensus layer that ensures mutually distrustful nodes agree on the same block history and the state layer that defines the state structure and transition rules.

Ethereum’s State. Ethereum maintains its state as so-called the world state [48]. As depicted in Figure 2, it is a mapping between the account address and the account state. An Ethereum account state consists of four fields: (1) balance, the number of Ether (the native cryptocurrency of Ethereum) owned by this account, (2) nonce, the number of transactions sent by this account, (3) code hash, the hash pointer of the smart contract code of this account, and (4) storage hash, the hash pointer of the root node of a Merkle Patricia tree [14] that

---

1http://github.com/XXXX/XXXX
encodes the key-value storage (the mapping between 256-bit integers) of this account.

Figure 2: Ethereum world state

The Ethereum’s world state transition rules are defined by the Ethereum Virtual Machine (EVM) [48], a quasi-Turing-complete machine. EVM is a 256-bit word stack machine with volatile byte-addressable memory and persistent key-value storage. The EVM code can access the stack via PUSH, POP, SWAP, DUP, etc., the memory via MLOAD, MSTORE, MSTORE8, etc., and the storage via SLOAD and SSTORE. When a transaction invokes a smart contract, the EVM executes the contract code and manipulates the world state. Each EVM execution step incurs a cost denominated in gas, and the transaction sender pays the total amount of gas consumed by the transaction.

Throughput Bottleneck. The throughput of a blockchain is determined by two factors: the block time and the block size. The block time is typically set as a constant at the consensus layer and has been considered the throughput bottleneck. For example, Proof-of-Work (PoW) consensus [36], introduced by Bitcoin, must keep a slow block generation rate (about 10 minutes a block) to avoid the generation of concurrent blocks. As a result, Bitcoin processes only about seven transactions per second. Nevertheless, in recent years, many consensus protocols which achieve fast block generation rates have been proposed and adopted in both industry and academia [30, 31, 44, 45]. For example, Ethereum has already switched to the efficient Proof-of-Stake (PoS) consensus [24], which enables short block time. Moreover, Conflux, a permissionless blockchain, can generate a block in less than 0.1 seconds [31]. With these advances, the consensus layer is no longer the performance bottleneck.

To further improve throughput, researchers then attempt to increase the block size. However, the block size cannot be arbitrarily large. Otherwise, the less performant full nodes would not catch up with the blockchain due to the storage space and execution speed requirements [12]. Sharding is a promising approach to address the space issue [9, 26, 33, 40, 47]. It splits the blockchain state into multiple shards so that every node holds only a portion of the overall state.

Nevertheless, sharding does not solve the speed issue due to the hot spot problem. Most transactions in a blockchain invoke only several highly-popular contracts and thus are likely to be processed by only several hot shards. That is, nodes in hot shards still have to execute a large number of transactions, which severely restricts the block size increment. Therefore, transaction execution acceleration is a crucial step toward high-throughput blockchains.

Limitation of Serial Execution. Most blockchain systems execute transactions within a block serially. Such serial transaction execution models are easy to reason about but fail to exploit the internal parallelism of modern commodity hardware (e.g., multi-core processors and SSDs). This motivates us to introduce concurrent transaction execution.

2.2. Concurrency Control Algorithm

Concurrency Control in DBMSs. Traditional DBMs give transactions the illusion that their execution is isolated from other transactions, i.e., they cannot observe the effects of concurrent transactions. Researchers have proposed various concurrency control algorithms to execute transactions concurrently while maintaining the illusion of isolation.

The core of a concurrency control algorithm is its strategy for handling conflicts. A conflict between two operations of different transactions occurs if they access the same object and at least one is a write. We classify existing concurrency control algorithms into two categories based on their conflict-handling strategies:

- **Pessimistic algorithms**, which use locks to avoid conflicts. They do not allow a transaction to access an object until the corresponding lock is acquired. The most frequently used pessimistic scheme is the two-phase locking (2PL) [10]. As its name suggests, 2PL divides transaction execution into two phases: (1) a growing phase where transactions only acquire locks on demand (i.e., no lock releasing), and (2) a shrinking phase where transactions only release locks (i.e., no lock acquiring).

- **Optimistic algorithms**, which assume conflicts between transactions are rare. They execute transactions concurrently without blocking any operation but abort and restart conflicting transactions once conflicts are detected. Optimistic concurrency control (OCC) [28] is one option. The execution of an OCC transaction consists of three phases: (1) a read phase, where the DBMS executes the transaction in its private workspace and tracks its read and write sets, (2) a validation phase, where the DBMS examines whether the transaction’s read set overlaps with other concurrent transactions’ write sets, and (3) a write phase, where the DBMS installs the transaction’s write set if the validation succeeds or otherwise aborts and restarts the transaction.

Concurrency Control in Blockchains. Unlike DBMs, blockchains impose an additional restriction on transaction execution — transactions must commit in the order specified
The most prominent characteristic of blockchain workloads is the **hot spot problem**. In most cases, ERC20 contracts lead to data conflicts when multiple transactions distribute tokens from the same sender. The function `transferFrom` (line 8) allows senders to transfer a certain number of tokens on behalf of owners. Tokens can be transferred via function `transferFrom`, where message senders transfer a certain number of tokens on behalf of owners. Tokens may be blocked for a long time or even aborted by other higher-priority transactions due to lock contention. For the optimistic algorithms, severe data conflicts lead to numerous aborted transactions. Therefore, **how to design an efficient concurrency control algorithm to deal with the hot spot problem in blockchain workloads** remains an open problem.

### 3.2. Toward Parallelizing an ERC20 Contract

#### An Example of Data Conflicts

Figure 4 presents a fragment of the implementation of an ERC20 contract. The contract’s persistent states are represented by two variables: the mapping `balances` that records the number of tokens owned by each account, and the mapping `allowances` that tracks the number of tokens authorized for third-party transfers by owners. Tokens can be transferred via function `transferFrom`, where message senders transfer a certain number of tokens on behalf of owners. The function fails if the owner does not have enough tokens (line 13) or the message sender does not have enough allowances (line 20).

In most cases, ERC20 contracts lead to data conflicts when multiple transactions distribute tokens from the same sender.
address [18]. For example, consider the following two concurrent transactions:

\[
\begin{align*}
\text{tx}_1 &= \text{transferFrom}_D(A, B, \text{value}_1) \\
\text{tx}_2 &= \text{transferFrom}_E(A, C, \text{value}_2)
\end{align*}
\]

where the function subscript denotes the transaction sender address (i.e., the return value of \_msgSender() in Figure 4). These two transactions conflict because they both read and write the same storage slot balances[A].

**Transaction-level Conflict Handling Strategy.** Traditional algorithms cannot parallelize \text{tx}_1 and \text{tx}_2 due to the conflict on balances[A]. Nevertheless, we find that most operations in both transactions do not depend on the conflict (i.e., balances[A]) and are thus conflict-free. We refer to such conflict-handling strategies as transaction-level strategies, because they view a transaction as a whole and are unaware of internal operations. When encountering a conflict, they either block or abort the entire transaction, which is inefficient under severe data contention.

**Operation-level Conflict Handling Strategy.** Let us consider the previous example. Under the optimistic concurrency control, \text{tx}_2 cannot pass the validation phase because it misses \text{tx}_1’s update on balances[A] and produces the wrong balances[A] (line 14 of Figure 4). But note that \text{tx}_2 does correctly update all other storage slots like balances[C] and allowances[A][E]. Inspired by the finding, we extend traditional OCC by introducing a redo phase after the validation phase. During the redo phase, rather than aborting \text{tx}_2 immediately after the validation fails, we re-execute line 14 based on the correct balances[A] value. In this way, we resolve the conflict on balances[A] by re-executing only one line of the source code and thus parallelize most operations in \text{tx}_1 and \text{tx}_2 successfully. We refer to this conflict-handling strategy as the operation-level strategy because it only re-executes operations that depend (directly and indirectly) on conflicts. Considering that most conflicts in blockchain workloads only influence several operations (as shown in this ERC20 contract example), the operation-level strategy can significantly alleviate the performance degradation caused by conflicts.

To guarantee the correctness of the operation-level strategy, we further introduce a technique called constraint guards. We use another scenario from the previous example to illustrate the necessity of the constraint guards. Suppose balances[A] does not own enough tokens to afford \text{tx}_2 after executing \text{tx}_1. In this case, \text{tx}_2 should be aborted due to the balance checking in line 13. Our operation-level strategy should detect the violation and abort \text{tx}_2, instead of simply re-executing line 14. To achieve this, when executing lines 13 and 20, we insert constraint guards that assert the required conditions must hold. During the redo phase of \text{tx}_2, we re-check these constraints, detect that the correct balances[A] does not satisfy the constraint in line 13, and thus abort \text{tx}_2.

### 4. Challenges and Our Solution

#### 4.1. Challenges

Our main idea is to resolve transaction conflicts at the operation level. We extend traditional OCC with a redo phase to identify and re-execute operations influenced by conflicts. Nevertheless, both the conflicting operation identification and re-execution are challenging at the EVM bytecode level.

**Conflicting Operations Identification.** To guarantee the correctness and efficiency of the redo phase, we must precisely identify all operations that depend on conflicts (i.e., without under-estimation or over-estimation). However, it is difficult to figure out the dependencies at the EVM bytecode level, as EVM operations do not state the data dependencies explicitly. In other words, they do not directly take the results of other operations as the inputs; instead, they retrieve inputs from the runtime context (i.e., stack, memory, and storage). This indirection makes it hard to identify the definition operations of these inputs.

**Conflicting Operations Re-execution.** The execution of an EVM operation depends on its runtime context. For example, to execute an \texttt{ADD} operation, we must know the top two elements in the stack; to execute a \texttt{MLOAD} operation, we must know the memory content at that point. However, it is difficult to obtain the runtime context corresponding to a specific operation during the redo phase without executing all previous operations. Without these run-time contexts, we cannot re-execute EVM operations correctly.

#### 4.2. Our Solution: SSA Operation Log

To address these challenges, we propose a new technique called SSA operation log, which records not only the operands and results of executed operations but also the data dependencies among these operations. The main idea is that instead of analyzing EVM bytecode which depends on the runtime context, we dynamically generate the static single assignment form (SSA) [5] representation of executed operations. More specifically, we enforce that the inputs of an operation must be explicitly represented by (i) immediate values, (ii) results of previous operations, and (iii) committed storage slots\(^3\). In this way, we eliminate all runtime context accesses, making the conflicting operation identification and re-execution much easier.

The SSA operation log provides well-defined use-definition and definition-use\(^4\) relationships among operations. During the redo phase, we can identify all conflicting operations by traversing the definition-use chains, re-construct operation inputs based on their definition operations, and finally re-execute these conflicting operations. For example, Figure 5 illustrates

---

\(^3\)We refer to a storage slot as committed if previous operations in the current transaction have not updated it.

\(^4\)When a variable, \texttt{v}, is on the left-hand side of an assignment statement, \texttt{s}, then \texttt{s} is a definition of \texttt{v}. When a variable, \texttt{v}, is on the right-hand side of a statement, \texttt{s}, then a definition of \texttt{v} is used at \texttt{s}.

---
We have implemented an open-source prototype, Olive, based on Go Ethereum (aka Geth) [11]. It is worth noting that the implementation of instruction inputs.

5.2. SSA Operation Log Generation

This section shows how to dynamically generate an SSA operation log in the read phase. An entry in the SSA operation log typically consists of the following fields:

- **LSN**, a unique log sequence number that identifies the entry.
- **Opcode**, the operation code of the operation.
- **Operands**, the inputs of the operation.
- **Result**, the output of the operation.
- **Def**, the definition operations of the inputs. This field comprises three sub-fields, namely stack, storage, and memory, which track the data dependencies via the stack, storage, and memory. We elaborate on the detailed structures and generation rules for them in Sections 5.2.1-5.2.3, respectively.

The fundamental principle behind the generation algorithm is to track the data flow and figure out the definition operations of instruction inputs.

5.2.1. Stack Operation Log Generation First, we present how to generate SSA operation log entries for operations that access only the stack. For each operation, Olive records the definition operations of all stack operands in the **def.stack** field. That is, if a stack operand, **operands[i]**, is the result of a previous operation, the **def.stack[i]** field is set to the **LSN** of that operation; otherwise, the **def.stack[i]** field is set to **NULL**.

To generate the **def.stack** field correctly, Olive employs a
technique called shadow stack, which records the definition operation of each stack item. If a stack item is the result of a previous log entry, Olive sets the corresponding shadow stack item to the LSN of that entry; otherwise, Olive sets the shadow stack item to NULL. In this way, for each operation, Olive can obtain the definition operations of its stack operands from the shadow stack and store them in the def.stack field.

Olive manipulates the shadow stack in a similar way as the stack. For the push operation, which pushes a constant data to the stack, Olive pushes NULL to the shadow stack. For the pop, swap or dup operation, Olive pops, swaps or duplicates the shadow stack. For other computational operations, Olive first obtains and pops all shadow stack items (denoted as lsn) corresponding to the stack operands. If lsn are all NULLs, which indicates that all operands are constants and thus the result is also a constant, Olive pushes NULL to the shadow stack. Otherwise, Olive generates a new log entry, stores the lsn in the def.stack field, and pushes the LSN of the entry to the shadow stack.

Figure 7 shows an example of generating a log entry for an ADD operation. Before executing the ADD, the stack contains two items (i.e., 30 and 20). According to the shadow stack, we can infer that the first item (i.e., 30) is the result of the eighth log entry, while the second item (i.e., 20) is a constant value. After the execution, Olive generates a new log entry for the ADD operation (i.e., the ninth entry), pushes the computation result (i.e., 50) to the stack, and finally pushes the LSN of the new entry (i.e., 9) to the shadow stack. In this way, the following operations can realize that the top stack item, 50, is the result of the ninth log entry.

For storage and memory operations, the generation rules for def.stack fields are similar. But they should additionally maintain the def.storage and def.memory fields.

5.2.2. Storage Operation Log Generation Then we consider storage operations, namely SLOAD and SSTORE. There are two types of SLOADs: (i) reading committed storage and (ii) reading values written by previous SSTOREs in the current transaction. Olive uses the def.storage field to distinguish them. For the first case, the def.storage field should be NULL, because such SLOADs do not depend on any previous operation; while for the second case, the def.storage field should be the LSN of the latest SSTORE entry that writes to the storage slot.

Olive maintains two maps to track the data flow over the storage. The first map, latest_writes, records the LSN of the latest SSTORE on each storage slot. The second map, first_reads, records the LSNs of SLOADs that read each committed storage slot directly. Olive uses the lastest_writes map to generate the SSA log entry for storage operations, while uses the first_reads map to identify conflicting storage operations. When generating a log entry for the SSTORE operation that writes to the storage slot A, Olive updates the lastest_writes[A] to the entry’s LSN. When generating a log entry for the SLOAD operation that reads from storage slot A, if A does not exist in the lastest_writes map, Olive sets the def.storage field to NULL and appends the entry’s LSN to the first_reads[A]; otherwise, Olive sets the def.storage field to the value of lastest_writes[A].

5.2.3. Memory Operation Log Generation In EVM, memory operations (e.g., MLOAD and MSTORE*) are not aligned. Therefore, multiple memory writes may overlap, and a single memory read may depend on multiple memory writes. Figure 8(a) shows an example of interleaved MSTORE and MSTORE8. In this case, the following memory operations that read these bytes must be aware that they depend on these two MSTORE* operations.

To track the data flow over the memory, Olive employs shadow memory, a technique similar to the shadow stack. As shown in Figure 8(b), for every byte set by a memory write, Olive set the corresponding data in the shadow stack to <LSN, offset>, where LSN denotes the log sequence number of that write operation and offset denotes the distance between the current byte and the first byte set by the memory write operation.

Moreover, Olive uses the def.memory field to record the data
dependencies for every memory read operation based on the shadow memory. Specifically, the *def.* memory field consists of multiple <start, len, lsn, offset> tuples, which denotes that the [start:start+len) bytes in the memory input are set by the [offset:offset+len) bytes in the result of the lsn-th log entry. For instance, if a MLOAD operation reads the 32-byte memory region in Figure 8(a), Olive will set the *def.* memory field to the tuples shown in Figure 8(c).

5.2.4. Constraint Guard Generation In order to guarantee correctness, Olive must satisfy certain constraints in the redo phase. First, the control-flow constraints state that the re-execution in the redo phase must follow the same path as the speculative execution in the read phase. For example, for a conditional jump (e.g., JUMP), the jump condition must remain the same. Similarly, for an unconditional jump (e.g., JUMP), the jump destination must not change. Therefore, when executing control-flow-related operations, Olive inserts constraints guards to ensure that control-flow-related operands (e.g., jump conditions and jump destinations) are unchanged. More specifically, if the shadow stack item corresponding to a control-flow-related operand is not NULL (i.e., the operand is not constant), Olive inserts a special ASSERT_EQ log entry whose operands field is the operand value and *def.* stack field is the shadow stack item. In this way, during the redo phase, Olive can compare the operands field and the result field of the log entry whose LSN is *def.* stack. If these two fields match, the constraint is satisfied; otherwise, the redo phase should fail.

Second, Olive also requires data flow constraints to ensure that the re-execution does not alter the data dependencies among operations. For example, for a MSTORE operation, Olive must ensure that its target address is unchanged in the redo phase. Otherwise, Olive cannot figure out which operation

Algorithm 1: The algorithm for the redo phase

```
Input: SSA operation log: oplog; definition-use graph: DUG; conflicting storage slot map: conflicts[keys]values.
Output: whether the redo phase is successful.
1 Function Redo(oplog, DUG, conflicts):
    2 sources ← SLOADs in oplog that read conflicts directly;
    3 for entry in sources do
        4 entry.result ← conflicts[entry.operands];
    5 end
    6 conflicting_ops ← DFS(DUG, sources);
    7 for entry in conflicting_ops / sources do
        8 if entry.opcode is ASSERT_EQ then
            9 expect ← entry.operands.value;
            10 current ← oplog[entry.operands.lsn].result;
            11 if current ≠ expect then return false;
        else
            12 reconstruct the inputs based on entry.def;
            13 re-execute entry and update entry.result;
        end
    end
    16 return true
```

depends on the address-changing MSTORE. To achieve this, if the target address of a run-time context access operation (e.g., MLOAD and MSTORE) is not constant, Olive inserts a ASSERT_EQ log entry to guard the address operand.

5.2.5. Definition-Use Graph Generation Based on the *def.* fields, Olive can find all operations that a specific operation depends on. Nevertheless, the conflicting operation identification in the redo phase requires a reverse procedure — find all operations depending on a specific operation. To achieve this, Olive constructs a definition-use graph, DUG, while generating the SSA operation log. Each log entry becomes a node in the graph, and each edge from the SSA operation log. Each log entry becomes a node in the graph, and each edge from
Table 1: Speedups achieved by different strategies. OCC+ denotes OCC with prefetching, and Olive+ denotes Olive with prefetching.

|                | Baseline | OCC  | Olive Prefetch | OCC+ | Olive+ |
|----------------|----------|------|----------------|------|--------|
| Speedup        | 1×        | 2.49× | 4.28×          | 2.89× | 3.25×  | 7.11× |

Figure 9: Speedup distribution for Olive

Figure 10: The correlation between block transaction number and average speedup

Otherwise, Olive re-constructs the operation inputs based on the results of all definition operations listed in the def field (line 13) and then re-executes the operation (line 14).

5.4. Implementation Details

We have implemented Olive in Go Ethereum (version 1.10.17). It consists of about 4200 lines of code changed: about 2600 for the optimistic concurrency control framework, and about 1600 for the operation-level conflict-handling strategy. Besides the design highlights presented above, Olive also resolves the balance and nonce conflicts at the operation level using similar techniques. Moreover, we implement Olive carefully and take into account corner cases of Ethereum, such as dynamic gas instructions and implicit gas limit checking.

6. Evaluation

In this section, we present a comprehensive evaluation of Olive on its correctness and performance. Our experiments are conducted on a platform with an Intel i7-10700F 2.9GHz processor (8 physical cores and 16 virtual cores), 16GB memory, and a 1TB SSD. The system software running on the platform is Ubuntu 22.04 LTS with Linux kernel 5.15 and Ext4 file system.

6.1. Correctness Validation

The correctness of Olive can be validated by the design of Ethereum itself. Ethereum maintains its state as a Merkle Patricia Trie (aka MPT) [14], where every non-leaf node contains the cryptographic hash of its child nodes. Therefore, two Ethereum states are identical if and only if the root nodes of their MPTs are identical. Moreover, for validation purposes, every Ethereum block contains the cryptographic hash of the MPT root node after all transactions in the block are correctly executed. We have run Olive to process the initial 11.5 million blocks from the Ethereum mainnet, and Olive always produced the MPT root nodes that matched the values in the block. The result demonstrates that Olive rigorously follows the rules defined in the Ethereum yellow paper [48].

6.2. Performance Evaluation

Speedup. To measure the performance of Olive, we run Olive to process 500k real-world Ethereum blocks (the block numbers range from 11 million to 11.5 million). Olive achieves an average of 4.28× speedup compared to Geth 1.10.17 (baseline). We show the distribution of speedups in Figure 9: most blocks are accelerated by 2~7×, only 0.88% are not accelerated, and 2.66% are accelerated by more than 10×. We also compare Olive with the traditional OCC. As shown in Table 1, OCC obtains only 2.49× speedup in real-world Ethereum workloads, which demonstrates that Olive significantly outperforms traditional concurrency control algorithms.

Besides, Figure 10 illustrates the correlation between the block transaction number and the speedup achieved by Olive. Olive generally obtains higher speedups on blocks that contain more transactions (i.e., with larger block sizes). It indicates that if the block sizes are enlarged in the future, which is the goal of transaction execution acceleration, Olive could achieve a higher speedup.

Scalability. Figure 11 shows the speedups of Olive and OCC as we increase the number of worker threads. Olive scales well to eight threads, while OCC only scales to four threads. The speedups of Olive increase slightly past eight threads, because there are only eight physical cores on our platform. More than eight worker threads would contend for cores, even the hyperthreading is enabled.

Speedup with Prefetching. Analyzing the runtime profiling
data, we identify that the performance bottleneck of Olive is storage operations, especially SLOADs. Accessing the persistent Ethereum state involves loading values from the on-disk LevelDB database, which incurs high latency. Previous works have proposed a state prefetching technique to address the problem [4]. On Ethereum, transactions are known to most participating nodes before they are packed into blocks by the miners. Therefore, to avoid the expensive disk read in the actual execution, one can pre-execute a pending transaction to predict which storage slots it may access and then prefetch and cache them in the memory. To evaluate how many speedups Olive can achieve if combined with the state prefetching technique, we modify the import functionality of Geth so that every block is processed twice. Specifically, we prefetch all required storage slots to the memory in the first execution and measure the performance of the second execution. As shown in Table 1, with the help of the prefetching technique, Olive further achieves an average speedup of \( 7.11 \times \). We report the speedup distribution in Figure 12. As comparisons, the prefetching technique itself and OCC with prefetching accelerate the execution by \( 2.89 \times \) and \( 3.25 \times \), respectively. These results suggest that Olive can cooperate better with the prefetching technique than traditional algorithms.

**SSA Operation Log.** To further illustrate the effectiveness of our operation-level conflict-handling strategy, we conduct detailed measurements on the SSA operation log generation and re-execution. Experiment results show that our SSA operation log generation algorithm only incurs an average of 4.5% runtime performance overhead for each transaction. Besides, our generation algorithm reduces the SSA operation log size significantly. The average EVM instruction count of contract invocation is 2559, while the average SSA operation log length is 127. In other words, the size of SSA operation log, including additional constraint guards, is only 5.0% of the original EVM instructions. This is mainly because our generation algorithm reduces the stack manipulation instructions (e.g., PUSH and POP) and instructions that do not depend on storage slots. Moreover, only seven log entries (0.3% of the original EVM instructions) on average are re-executed in the redo phase, and the time spent on the redo phase is only 4.9% of the overall block processing time. These results demonstrate the efficiency of the redo phase.

**Resource Utilization.** Finally, we report the resource utilization of current implementation of Olive compared to the official Geth (v1.10.17). On our platform, the average memory consumption of Olive is 9.48 GB, and the average utilization of 16 vCPUs is 32.19%. In contrast, the official Geth consumes 9.08 GB memory and utilize 19.38% of all CPU resources. In other words, Olive incurs only 4.41% memory overhead (caused by the shadow stack and shadow memory) while increases the overall CPU utilization by 66.10% (thanks to the parallel transaction execution). The overall CPU utilization of Olive is not increased dramatically, because Olive only parallelize the transaction execution component of Geth. Many other components, like the expensive state committing, are still single-threaded.

**7. Related Work**

Existing works for improving the throughput of blockchains can be broadly categorized into two classes. The first class aims to enlarge the block size by accelerating the transaction execution, while the second class seeks to shorten the block time by enabling faster block generation rates.

**Transaction Execution Acceleration.** The concurrent execution of blockchain transactions has become the focus of much research in recent years. Saraph et al. [41] proposed a parallel execution engine, where they execute all transactions in parallel, roll back transactions that cause data conflicts and then re-execute these conflicting transactions sequentially. They also observed that several popular contracts caused severe contention and performance degradation. Garamvölgyi et al. [18] conducted an empirical study on the parallelism of the Ethereum workloads and identified that most data conflicts derive from counters (e.g., balances). They further introduced partitioned counters and special commutative instructions to alleviate data conflicts. Dickerson et al. [8] described a parallel smart contract execution approach for miners and validators based on techniques adapted from software transactional memory [6, 21, 22]. They allow miners to execute smart contracts in parallel and resolve data conflicts by delaying or rolling back some conflicting invocations. Besides, miners also generate concurrent schedules dynamically so that validators can convert the schedules into deterministic, parallel fork-join programs. Similarly, Anjana et al. [2] harness optimistic software transactional memory systems to execute transactions concurrently in miners. They also introduce block graphs to capture the dependency of transactions in miners and allow validators to re-execute transactions concurrently and deterministically based on the graphs. Zhang et al. [51] employed the multi-version timestamp ordering (MVTO) scheme to reduce transaction aborts in the optimistic concurrency control. Jin et al. [23] proposed a concurrency protocol that focuses on the validator-side concurrency in permissioned blockchains. They divide the transaction dependency graph generated by miners into several sub-graphs to preserve parallelism and reduce communication costs. They further integrated their concurrency control algo-
algorithm with the PBFT consensus protocol. Nevertheless, all these works use traditional transaction-level conflict-handling strategies. Therefore, in real-world blockchain workloads, they encounter frequent delays or aborts and thus suffer severe performance degradation. Olive, instead, uses an operation-level strategy where most operations, even in conflicting transactions, can be executed concurrently without blocking or aborting. Speculative execution is another approach to speed up transaction execution [3, 4, 20, 27, 37]. It utilizes the time window between when a transaction is known and when it is executed to predict and pre-execute transactions speculatively. Forerunner [4] proposed a constraint-based approach for speculative pre-execution. It speculates on multiple futures and accelerates transactions based on imperfect predictions whenever certain constraints are satisfied. It is worth noting that Forerunner is complementary to Olive: Forerunner focuses on the pre-execution phase, while Olive concentrates on the execution phase. We have shown that Olive can cooperate well with the state prefetching techniques. Therefore, we believe that Olive can achieve even better speedup if combined with more sophisticated speculative execution approaches such as Forerunner.

**Block Generation Acceleration.** To reduce the consensus time between consecutive blocks, researchers have proposed various consensus algorithms based on Byzantine fault tolerance (BFT) protocols [19, 25, 34, 35, 38]. Besides, to replace the longest chain rule used by PoW, researcher have explored alternative structures to organize blocks, such as the DAG-based chains [30, 31, 42–44], hierarchical chains [16, 39] and parallel chains [49]. Although these techniques achieve higher block generate rates, most of them suffer scalability issues or live attacks [32]. In contrast, Olive can accelerate the transaction execution without sacrificing security and have full backward compatibility. Moreover, Olive is orthogonal to these techniques. Therefore, the overall throughput of blockchains can be multiplied if Olive is combined with these approaches properly.

Sharding is a popular solution to address the scalability issue of blockchains [1, 7, 9, 26, 29, 33, 40, 46, 47, 50]. Sharding systems split the blockchain states into multiple shards and select a small committee to process transactions on each shard. Nevertheless, they also suffer security issues due to the slow committee reconfiguration. Moreover, sharding exacerbates the hot spot problem in blockchain workloads: most popular contracts invocations are likely to be processed in several hot shards, and transactions on each shard are more likely to conflict. Olive can help sharding systems resolve these conflicts at the operation level and mitigate the hot spot problem.

### 8. Conclusion

In this work, we present, implement and evaluate Olive, an operation-level concurrent transaction execution system for Ethereum. We resolve data conflicts by identifying and re-executing conflicting operations, so that all other conflict-free operations can be executed concurrently without blocking or aborting. Evaluation results demonstrate that Olive can effectively accelerate transaction execution in real-world Ethereum workloads.

### References

[1] Mustafa Al-Bassam, Alberto Somnino, Shehar Bano, Dave Hryczyszyn, and George Danezis. Chainspace: A shared smart contracts platform. arXiv preprint arXiv:1708.03778, 2017.

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

[3] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M. Frans Kaashoek, and Nikoali Zeldovich. Efficiently migrating transient execution attacks using the unmapped speculation contract. In Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, OSDI’20, USA, 2020. USENIX Association.

[4] Yang Chen, Zhongxin Guo, Runhuai Li, Shao Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative execution for ethereum. In SOSP 2021. Association for Computing Machinery, October 2021.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4):451–490, oct 1991.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, page 336–346, New York, NY, USA, 2006. Association for Computing Machinery.

[7] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In Proceedings of the 2019 International Conference on Management of Data, SIGMOD ’19, page 303–312, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC ’17, page 302–312, New York, NY, USA, 2017. Association for Computing Machinery.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. Commun. ACM, 19(11):624–633, nov 1976.

[10] Ethereum. Go ethereum. https://github.com/ethereum/go-ethereum.

[11] Ethereum. Blocks. https://ethereum.org/en/developers/docs/blocks/, 2022. accessed: 2022-08-30.

[12] Ethereum. Introduction to smart contracts. https://ethereum.org/en/smart-contracts, 2022. accessed: 2022-08-30.

[13] Ethereum. Patricia merkle trees. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-tree/, 2022. accessed: 2022-08-30.

[14] Ethereum. Solidity documentation. 2022.

[15] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. [Bitcoin-NG]: A scalable blockchain protocol. In 13th USENIX symposium on networked systems design and implementation (NSDI 16), pages 45–59, 2016.

[16] Vitalik Buterin Fabian Vogelsteller. Eip-20: Token standard. https://eips.ethereum.org/EIPS/eip-20, 2015. accessed: 2022-08-30.

[17] Peter Garmavölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In Proceedings of the 44th International Conference on Software Engineering. ACM, may 2022.

[18] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP ’17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[19]
