Parallel Transaction Execution in Public Blockchain Systems

by

Rizwan Shahid

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Public blockchain systems like Ethereum and Bitcoin suffer from poor transaction throughput, leading to delayed transaction execution and high transaction fees. They execute transactions one by one, failing to extract inherent parallelism possible in executing the workload.

We present Block-X, a parallel transaction processing system with a serializable concurrency control that executes transactions in a block in a serializable order equivalent to the order of transactions in the block for public blockchains. It pre-executes transactions that are waiting to be added to a block. Through this pre-execution, Block-X estimates the keys a transaction wants to read or write. It uses this information to create a parallel execution schedule and run transactions optimistically in parallel following the schedule. It also uses the pre-execution to prefetch data that will be accessed during the critical path transaction execution. If a smart contract transaction accesses data outside of its initially estimated read-write set of keys, Block-X detects and resolves any potential conflicts. The final state is equivalent to the state produced after the sequential execution of transactions in the block order. Finally, Block-X also accelerates the process of validating blocks by providing the parallel execution schedule produced in the block execution step to validate transactions in parallel.

We implemented our system on Ethereum so it is compatible with EVM chains. Our evaluation demonstrates that Block-X achieves up to a 2.3× higher throughput than Ethereum. Moreover, our performance is comparable to other systems that perform pessimistic execution. These systems require predefined read-write set and reject transactions that use data outside of it.

Acknowledgments

I would like to thank my advisors, Bernard Wong and Samer Al-Kiswany, who let me explore problems and eventually work on this one with great flexibility. I enjoyed great freedom in every aspect of working with them. Their unwavering support and guidance were instrumental to make this possible. Fueled by curiosity and excitement, I think I ventured too far at times. I wasn't afraid to step outside of our area of expertise to ask if we can do better. They would sometimes steer me back, reminding me of our primary goal. Additionally, I also thank the readers of my thesis, Trevor Brown and Khuzaima Daudjee, for their detailed and insightful comments.

The thesis wouldn't be possible without the love and support of my family and friends. Their encouragement and prayers were a constant source of motivation, pushing me forward even in the face of challenges. I am grateful to my friends for listening to my ideas and providing valuable feedback. I enjoyed the thoughtful conversations we had, whether in the research lab, at restaurants, or around the pool table. The jokes we shared and the arguments we had about blockchains and cryptocurrencies were so much fun. I will forever cherish these memories, as they have enriched my graduate experience in ways that are truly invaluable. To each of you who has been a part of this journey: you know who you are, and I thank you for being that person.

And, of course, my gratitude extends to Nakamoto and Vitalik, whose work kept me awake at night, driving my exploration and learning in the blockchain world. I still feel there is a lot more to learn.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smart contracts are increasingly used to build a wide range of applications in fields such as finance [35, 43, 34, 9], IoT [17], healthcare [25, 12], and supply chain [36, 33]. Centralized services like VISA has the capacity to process 65000 transactions per second[41]. However, public blockchain systems offer poor transaction throughput. They struggle to cope with the increased demand because of their low throughput [7, 18] that leads to significant increases in transaction processing fees [31]. To achieve mainstream adoption of emerging smart contract applications, modern blockchain systems must offer high throughput for processing transactions.

Two main factors impact the performance of state-of-the-art blockchains: slow consensus protocols that agree on new state of blockchain [13], and inefficient processing of smart contract transactions. Recent efforts focused on improving the performance of the consensus protocol [24, 45, 20, 37] leading to the adoption of new protocols in public blockchains [24, 15]. However, there is relatively limited work on improving the throughput of transaction processing. State-of-the-art blockchains have a low throughput as they process transactions sequentially and fail to utilize inherent concurrency in the workload.

Previous work tried to accelerate transaction execution by executing multiple transactions in parallel using one of the following approaches: optimistic and pessimistic concurrency control. Systems that use optimistic concurrency protocol [23, 5, 30] execute transactions in parallel and subsequently detect conflicts. If a conflict is detected, transactions are aborted and re-executed. In public blockchain workloads, transactions perform expensive cryptographic operations and have a long chain of dependencies. These systems suffer from high abort rate, resulting in multiple re-executions. Systems that follow the pessimistic concurrency control [40, 44] require knowing the set of keys a transaction will

read or write in advance. This scheme locks accessed keys before the execution of a transaction to avoid conflicts. Any transaction accessing data outside of the set is aborted and rejected. This method complicates application development, especially with transactions that change the set of accessed keys at runtime.

We present Block-X: a high-throughput transaction processing system that uses a hybrid approach for concurrency control. Block-X uses read and write sets for each transaction to build a dependency graph and generate initial parallel execution schedule. To avoid tasking the developers with providing these sets, Block-X pre-executes transactions to estimate their read-write sets. To support transactions with read-write sets that change between runs, Block-X detects new dependencies at runtime and fall back to serial execution of conflicted transactions. This approach overcomes the limitations of the current systems. Pre-executing transactions relieves developers from setting read-write set as needed by the pessimistic approach. Using read-write set as hints to build an execution schedule, reduces the number of aborts experienced by the optimistic approach.

To achieve high throughput, Block-X leverages three workload characteristics. First, most transactions are known in advance [16]. They are buffered in memory (a.k.a, mempool) until they are included in a block. Block-X pre-executes transactions while they are waiting in the mempool. Pre-execution has two benefits. It identifies the read-write set for a transaction. Block-X uses this set to build dependency graph and generate an initial parallel execution schedule for transaction processing in a block. Furthermore, pre-execution fetches and caches data from the disk. This significantly reduces I/O overhead of the execution of transactions in the critical path.

Second, transactions rarely alter their read-write set between re-executions. Block-X implements an optimized path for the majority of transactions that only accesses keys in their read-write set and support two policies for handling conflicts: *Discard-and-reexecute* and *merge-and-resume*. In the Discard-and-reexecute approach, we discard intermediate state mutations, merge the set of transactions of the two conflicted worker threads, and re-execute the merged set of transactions serially. In the merge-and-resume approach, we merge the set of transactions of the two conflicted threads, merge the mutually disjoint intermediate state mutations, re-execute the conflicted transactions, and execute the rest of the merged transactions serially. Discard-and-reexecute is simple and has low memory footprint, whereas merge-and-resume has better block execution time.

Third, validators repeat the work done by miners to validate a block. Block-X leverages this insight through recording the parallel execution schedule used by miners and making it available to validators. We use the recorded schedule at the validators to execute the transactions in parallel without facing any conflicts. Validators consider block as invalid if

they detect a conflict. This is because they may observe a state that is inconsistent with the one observed when the block is executed sequentially.

To evaluate the benefits of the proposed approach we build a prototype of Block-X over the Ethereum Geth go-based client. We compare the performance of Block-X to sequential execution in Etheruem and pessimistic parallel approach. We use 3000 blocks from Ethereum blockchain. Our results show that Block-X achieves up to a 2.3× higher throughput compared to sequential execution of Etheruem. In our workload, the pessimistic approach rejects 4% of the transactions. Block-X achieves comparable performance to the pessimistic approach without rejecting any transactions.

Chapter 2

Related Work

Previous efforts to execute transactions in parallel followed one of the following schemes:

Optimistic concurrency control. In this scheme transactions are executed optimistically and are subsequently validated to detect conflicts. If a conflict is detected transactions are aborted and re-executed. In public blockchains with long chains of dependant transactions, this scheme has a high abort rate leading to multiple re-executions.

Block-STM [23] is an in-memory, multi-version, optimistic parallel execution engine. Block-STM executes transactions concurrently; if a conflict arises, Block-STM re-executes the transactions with the new knowledge of dependency. It runs this trial-and-error cycle until the execution is successful. This approach works well under low conflict. If there are long chains of dependencies it can result in a high abort rate and can take more time than the serial execution.

Monad [5] optimistically executes transactions in parallel, then merges the updated state serially to check for conflicts. It avoids optimistic trial and error approach by using static code analysis to identify dependencies. Static code analyzer may impose a high overhead due to cross-contract calls and often over estimate the keys in the read/write set. This overestimation reduces the chance for running transactions concurrently.

Pessimistic concurrency control. This scheme requires knowing the set of keys a transaction will read or write. It locks accessed keys before the execution of a transaction to avoid conflicts. This scheme tasks developers to provide accurate read-write set in advance. Any transaction accessing data outside of the set is aborted.

Sui [40] is a blockchain that uses pessimistic concurrency control. Sui defines three types of objects: owned, immutable, and shared. Owned objects are accessed by the owner

only. Immutable objects can be accessed by multiple threads but cannot be mutated. Shared objects can be mutated by anyone. Sui requires developers to identify the shared objects a transaction will access [21]. Sui uses this information to optimize consensus as well as transaction processing. Transactions that do not have conflict on shared objects are executed concurrently.

Sei v1 [30] blockchain uses pessimistic concurrency control. It handles conflict by maintaining a mapping of transaction dependencies such that only non-conflicting transactions are allowed to run in parallel. It requires smart contract developers to define their resource dependencies. If contract dependencies are incorrectly defined the execution of a smart contract fails and a higher fee is charged. Sei v2 [38] is now adapting optimistic approach. Similar to other approaches, it runs transactions optimistically and handles conflicts by rerunning them sequentially until all conflicts are resolved. At the time of writing, Sei is still working on the v2 implementation and are yet to roll out optimistic parallel execution framework.

Similarly, Solana [44] requires identifying the set of keys a transaction will access. If a transaction attempts to access keys outside the identified set, Solana rejects the transaction.

Unlike previous approaches, Block-X follows a hybrid approach. Block-X does not task developers to provide strict read-write set for a transaction, alternatively, it pre-executes transactions to predict the keys they access. It uses this information to build a parallel execution schedule similar to pessimistic concurrency control. For transactions that access keys outside their identified read-write set, Block-X provide an efficient way to handle conflicts similar to optimistic concurrency control. It uses the predicted read-write set to minimise aborts in an optimistic execution of transactions.

Chapter 3

Background

Blockchain systems embody a novel computing paradigm with decentralized trust at its core. They implement replicated state machine that enables multiple entities, such as miners and validators, that secure the network to agree on the state changes. We now present a background on transaction execution in the Ethereum block chain. We notice that other blockchains have a similar architecture.

Ethereum is a popular public blockchain that supports Turing complete smart contracts. Smart contracts are computer programs that run on the blockchain. Smart contracts may have arbitrarily complex logic. Clients that run the blockchain software are called nodes. They form the backbone of the blockchain network. Two types of nodes play a critical role in a transaction execution in Ethereum: miners and validators. Clients submit transactions to miners. A miner validates the transaction, adds the transaction to its pool of transactions (a.k.a, mempool), and then broadcasts the transactions to other miners. Other miners consequently validate the transactions and add them to their mempool.

Miners select a set of transactions from the mempool to include in the next block, execute these transactions, and then participate in the consensus protocol to commit the block on the chain. If a miner is selected by the consensus protocol, it adds the block to the chain.

A subset of ethereum nodes act as validators. A validator validates newly added blocks on the chain. They re-execute the transactions in the new block and verify the state changes.

Executing transactions in a new block, reaching consensus, and validating a new block are all on the performance critical path. Blockchain nodes start working on the next block

Object	Description
Address	20 byte long account address
Balance	Account balance
Nonce	Account transactions processed
Code Hash	Hash of the contract code
Storage	Persistent memory

Table 3.1: Some attributes of Ethereum's state object

once the current block is validated. This is because the current block may modify the state that is required by the transactions in the next block. These three steps impact the throughput of the system. Previous efforts [24, 45, 20] investigate techniques to improve the consensus protocol. In this work, we focus on improving the throughput of the execution step at the miners and the validation step at validators.

The Ethereum Virtual Machine (EVM), a stack-based virtual machine, is responsible for executing transactions. It uses a stack to store instruction input and output. Given a state input to a transaction, it produces a deterministic output following the rules specified in [43]. The EVM supports smart contract execution, which is a program on the blockchain that comprises executable instructions. An instruction may read data from the state. Each instruction has an associated amount of gas. Users pay gas fee for their transactions to be included in the block. During the execution, EVM runs the instruction and burns the associated amount of gas to prevent the execution from running into an infinite loop.

Ethereum has an account based storage model. Both user and smart contract accounts use same object representation. Table 3.1 lists some attributes of the object. In addition to the user object attributes smart contract objects have storage associated with it. The storage is a persistent read-write memory. It is organized as a Merkle patricia trie [4]. Each object has upto 2^{256} slots, each 32 bytes long, directly accessible using indexes starting from 0. State variables defined in contracts are mapped to slots in the storage. Statically sized data is stored contiguously one after the other in slots starting at index 0. However, maps and dynamic arrays have unpredictable sizes, preventing them from being stored among statically sized data. Elements in these data structures are stored at locations determined by computing a hash. For instance, if a one-dimensional data array A is assigned to slot X, then $A[0]$ is stored at $keccak256Hash(X) + 0$. We utilize the storage index values to identify the state access.

Chapter 4

System Design

Popular public blockchain systems like Ethereum execute transactions sequentially limiting their transaction throughput. They under-utilize multi-core capabilities of modern hardware. Block-X aims to execute block transactions concurrently such that the final state is equivalent to that of sequential execution in block order. Our architecture design focuses on extracting parallelism in the execution of transactions by miners and validators.

In order to execute transactions in parallel, we pre-execute transactions in the mempool to estimate their read-write sets. In the critical path, miners sample transactions from the mempool and place them in a particular order in a block. Each transaction is assigned a unique index called T_{ind} and a total ordering is provided among transactions in a block. The total ordering of all transactions in a block that is created using the T_{ind} indicates the block order.

The estimated read-write sets are then used to generate a dependency graph of the transactions. Block miners follow the dependency graph to schedule independent transactions to different worker threads. Due to the Turing complete nature of smart contracts, transactions can access data outside of the previously estimated read-write set. Executing these transactions in parallel can potentially lead to unexpected data dependencies which if left unaddressed can result in a state that is different from the state generated after sequential execution of the transactions. Block-X introduces two approaches to handle such conflicts: 1) Discard-and-re-execute 2) Merge-and-resume. Both re-execute necessary transactions but differ in how they handle intermediate state changes. Once a miner generates a block, it provides the dependency information of the transactions in the block to other validators. Validators can use that information to validate the block in parallel.

Block-X builds on top of the Go-Ethereum client. Figure 4.1 shows its high level system

architecture. Block-X adds or modifies (colored in grey) the following components to Go-Ethereum: Pre-execution, Dependency builder, Scheduler, Worker threads, State Storage, and Conflict-handler. We will describe these components in turn in the following sections.

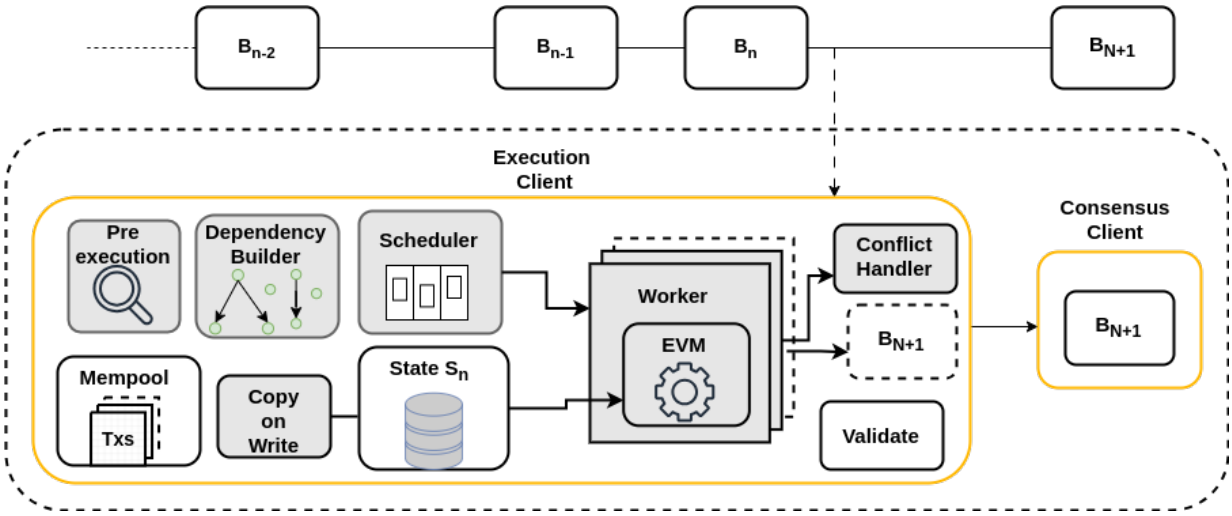

Figure 4.1: Block-X components. Modified/Added components are colored in grey

4.1 Transaction pre-execution

In order to safely execute transactions in parallel, we need to find dependencies between transactions. We can do this by analyzing the read-write keys generated from executing transactions before they are included in a block. Transactions are known well in advance [16] and are buffered in the mempool as shown in Figure 4.2. Transaction pre-execution gives an estimate of read-write keys because all executions are performed on the same state snapshot before the block order is created. A transaction execution in a block order may have a different control flow and data dependencies resulting in a different read-write set. Block-X ensures all data accesses outside of the estimated read-write set are safe.

Transactions in Ethereum consists of simple transfer transactions and complex smart contract transactions. A transfer transaction needs two account keys that do not change when it is executed with different block order. These values can easily be extracted from the transaction's plain text. However, smart contract transactions can have arbitrarily complex logic that can depend on input data and block metadata such as hash and timestamp that is not known until the block is created. Blockchain state data that is used as an input to

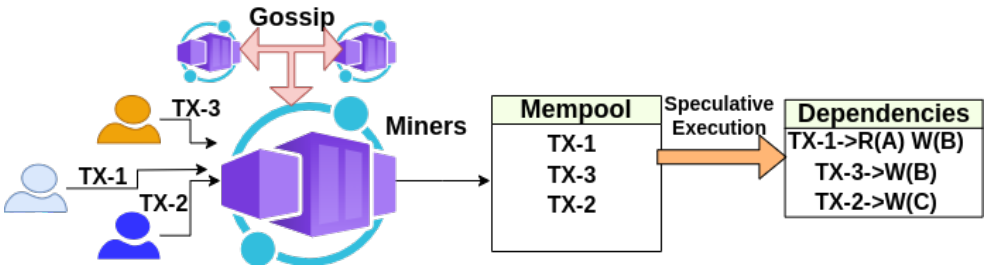

Figure 4.2: Transaction pre-execution in mempool

a smart contract may have different value due to a prior transaction executions in a block. With different values as input, a smart contract program may have a different control flow producing different state output.

In the Ethereum client, we first take a snapshot of the latest state, then optimistically execute a transaction while logging all data reads and writes to generate a list of all accesses performed by the transaction. We inspected EVM opcodes and identify those that read or write data from the state. They are listed in Table 4.1. The logger only needs to log keys whenever these specific EVM opcodes are called during the execution process. The pre-execution of a transaction is capped using a "gas" limit to avoid running into an infinite loop. After the execution, Block-X keeps the access-list but discards the state changes that are stored in the local buffer.

Overall, the pre-execution step can double the transaction execution in the worst case. However, in Ethereum, miners start processing blocks even before their turn. They discard execution and update their state as soon as they see a new valid block. In proof of work chains, miners are continuously processing blocks and competing to add their block to the chain. Only the winner gets to add its block, other miners simply discard their work

EVM Instruction	**Operation**
SLOAD	Read
SSTORE	Write
EXTCODECOPY, EXTCODEHASH, EXTCODESIZE	Read
BALANCE, SELFBALANCE	Read
SELFDESTRUCT, CREATE	Write
DELEGATECALL, CALL, STATICCALL, CALLCODE	Read

Table 4.1: Classification of EVM instructions by operation type

and start mining from the new updated state. This work can be used as an estimate of read-write keys for the subsequent execution of transaction. Read-write set can also be estimated using static analysis of the smart contract. However, static analysis tend to over estimate the read-write keys and can be computationally expensive if exponential contract code paths are explored.

Notation	Description
T	A set of transactions
readSet(t_i)	The read keys of t_i
writeSet(t_i)	The write keys of t_i
accessSet(t_i)	readSet(t_i) $\cup$ writeSet(t_i)

Table 4.2: Notation Table

4.2 Dependency builder

Typical OCC-based concurrency control protocols assume there is low contention [28] in the workload. However, this assumption does not hold true for Ethereum which we will explain in Section 5.1. Ethereum's mainnet traffic exhibits high contention because a few smart contract applications are popular. OCC protocols suffer from high aborts under high contention. Moreover, if they provide strict serializability it further adds overhead and restrict concurrency. Aborts are expensive as they incur significant resource overhead and require additional resources and time to revert transactional changes. Smart contract transactions, in particular, perform expensive cryptographic operations such as hashing and may result in large state changes. Therefore, one of our primary design objectives is to minimize runtime aborts, and Block-X achieves this by leveraging a dependency graph.

Definition 1: Transactional Dependency A transactional dependency where t_j depends on t_i (denoted as $t_j \rightarrow t_i$) *if and only if* $j > i$ and

$$\begin{aligned} & \text{writeSet}(t_i) \cap \text{accessSet}(t_j) \neq \emptyset \\ \text{or} \quad & \text{accessSet}(t_i) \cap \text{writeSet}(t_j) \neq \emptyset \end{aligned}$$

The transactional dependency incorporates three types of dependencies $\forall j > i$:

1. Write-After-Write (WW): t_j WW depends on t_i if t_j tries to update a key that is written by t_i.

2. Write-After-Read (WR): t_j WR depends on t_i if t_j tries to update a key that is read by t_i.

3. Read-After-Write (RW): t_j RW depends on t_i if t_j tries to read a key that is updated by t_i.

Definition 2: Dependency Graph Given a set of transactions $T = \{t_1, t_2, ...t_n\}$, the dependency graph $G(V, E)$ is a directed graph where vertices $V = T$ and edges $E = \{(t_i, t_j) \mid t_i \rightarrow t_j\}\ \forall t_i, t_j \in T$.

The dependency builder uses the read-write sets of transactions, determined during the pre-execution step, to build the dependency graph. The dependency graph is used by Block-X to find the weakly connected components(referred to as connected components for brevity) in the graph. The scheduler uses that information to create a parallel transaction execution schedule prior to the execution step. By incorporating early dependency information in the execution schedule, Block-X can reduce expensive runtime aborts that can slow down the system.

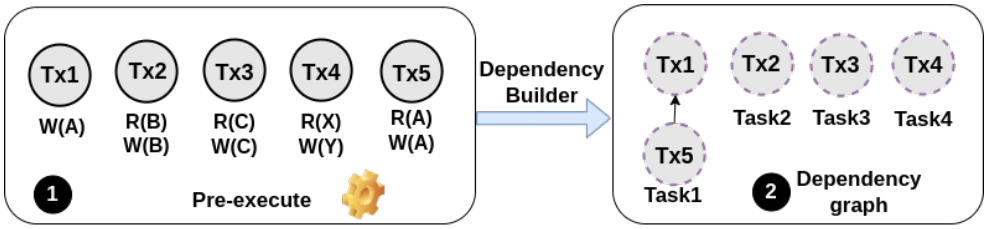

Figure 4.3: An example of dependency graph built using read-write estimates from pre-execution

Figure 4.3 shows an example of a dependency graph. In this example, the block consists of 5 transactions. Transactions are ordered according to T_{ind}: $[T_1, T_2, T_3, T_4, T_5]$. Tx1 is estimated to write object A. Tx5 is estimated to read object A before writing to it, creating a RW dependency edge: $t_5 \rightarrow t_1$. Other transactions are estimated to be independent, without any dependency edge. Note that during the critical path execution, transactions may access data outside of the estimated read-write set creating a conflict.

4.3 Scheduler

The scheduler is responsible for assigning tasks, consisting of a subset of transactions within a block, to worker threads for execution. The scheduler has a pool of idle worker threads

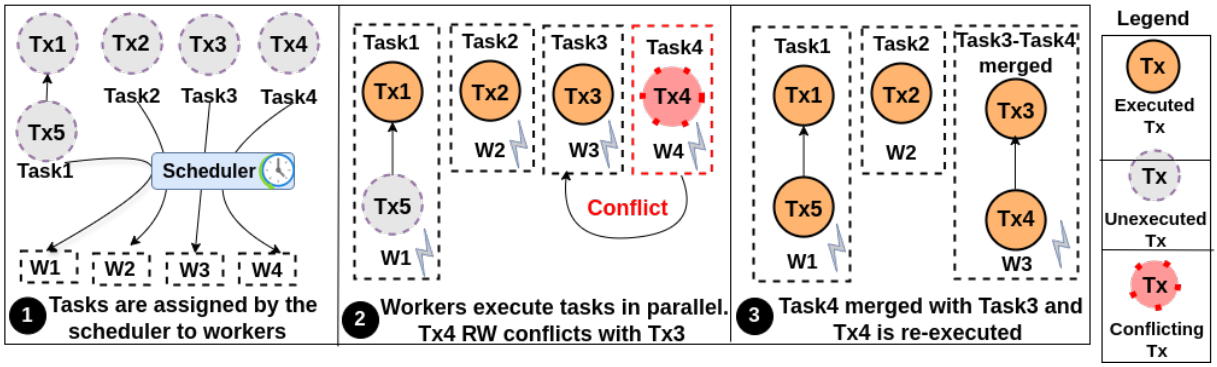

Figure 4.4: Example of parallel execution in Block-X. (1) The scheduler assigns tasks, comprising of set of transactions, to worker threads. (2) Worker threads execute tasks in parallel. Tx4 failed to observe an updated object written by Tx3, creating a RW conflict. (3) The conflict is resolved by merging the tasks and re-executing Tx4 using the updated object.

available for execution. It begins by creating tasks using the connected components in the dependency graph and then initiates the execution phase of the transactions by assigning the tasks to the first available thread in the pool. Worker threads execute tasks in parallel before returning back to the thread pool.

Every new task, created using the parallel schedule, starts with the *ready* status. When the scheduler assigns the task for an execution to a worker thread from the pool, it sets the task status to *running*. If all transactions successfully complete execution, the task is marked *executed*. However, if a conflict arises during the execution of a task, the status changes to *waiting-to-merge* where the conflict handler waits for the conflicted task's arrival so that the merge operation can be performed to resolve the conflict. After the merge, the task is *ready* again for re-execution.

Part 1 of Figure 4.4 shows an example in which scheduler assigns four tasks to four different workers. Task-1 consists of two transaction: Tx1 and Tx5. In the dependency graph, a path exists from Tx5 to Tx1; hence they are part of one connected component that is assigned to worker-1. In the second step, workers execute tasks in parallel. Tx4 RW-conflicts with Tx3, creating a conflict that is resolved by merging Task3 with Task4. Finally, in step 3, the merged task is again re-executed by worker-3.

The scheduler waits for all the worker threads to execute all the tasks before sending them to batch commit the state updates that are buffered locally. This approach has both advantages and disadvantages. On one hand, writes in the dependent transactions overlap,

for instance Tx5 in Figure 4.4 writes to the same object as Tx1; and hence batching the writes reduces commit operations. Moreover, data reads of subsequent dependent transactions in a task can be serviced from the local buffer of a worker thread instead of shared memory or database. On the other hand, it delays a transaction's write visibility to other worker threads until the final batch commit operation. For instance, Tx4 in Figure 4.4 failed to observe Tx3's write. Stale reads may arise if multiple worker threads try to read and then update the same data without seeing prior writes. Therefore, the scheduler services all dependent transactions to one worker thread so that dependent transactions see the latest state according to the block order. A worker thread executes task transactions following the block order and locally buffer all the state changes. Finally, as we will now describe, a commit operation moves all the buffered changes to the global state.

The Commit phase is separated from the execution phase with a barrier. Before the commit phase begins, all transactions in the block must successfully finish execution. The commit phase also requires all threads to synchronize before the scheduler sends all tasks to commit. The commit operation can be run on multiple threads, moving the locally buffered changes from successful task executions to the global state. The commit step can happen in parallel because locally buffered state updates do not overlap. The conflict handler ensures that only non-conflicting tasks successfully finish execution in parallel, meaning all transactions within one successfully executed task have no dependencies on any transactions from other tasks. If a system failure occurs during the commit, the block can be re-executed deterministically to produce the same state that can be persisted to the database.

The scheduler uses the block transactions to create a parallel schedule. For all block transactions, it uses the dependency graph to find all the connected components. An individual connected component encompasses all dependencies within the component transactions. All transactions in the components are executed sequentially in block order by the worker threads. However, sequential execution of a connected component may not fully leverage parallel execution opportunities among the transactions. For instance, consider three transactions t_1, t_2, and t_3 with dependencies $t_2 \rightarrow t_1$ and $t_3 \rightarrow t_1$. All transactions belong to a single connected component. If the entire component is included in one block, the three transactions will be executed sequentially, followed by a single batch commit step. Alternatively, the transactions can be divided into two blocks: the first block consisting of t_1 and the second consisting of t_2 and t_3. Now, t_2 and t_3 can be executed concurrently by two separate workers; however, this requires an additional commit step. Breaking transactions into smaller sets breaks the dependency graph components into smaller components reducing the transaction dependencies and increasing the parallelism. It comes at the cost of extra commit step. Given that Ethereum's workload consists of long chain of depen-

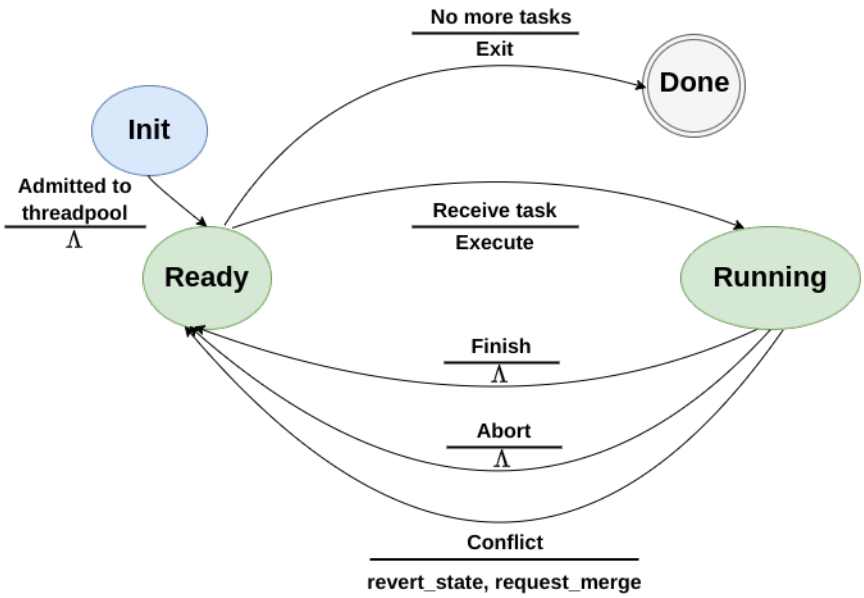

Figure 4.5: State transition of a worker thread. An arrow indicates the state transition. The event that causes the transition is shown above the horizontal line while the actions performed due to the event are shown below the horizontal line. Λ indicates null.

dencies (as further explained in Section 5) and it exhibits significant commit overhead, we choose not to divide block transactions into smaller sets for execution. This minimizes thread synchronization and commit overhead.

4.4 Worker thread logic

Block-X is designed to avoid synchronization between threads that can create unnecessary delays. A worker thread is responsible for executing a task assigned by the scheduler. Only one worker thread handles a task at a time. Each worker thread has access to the latest global state that it uses to read data. Data is kept in the thread-local buffer once it is retrieved from the global state. After the scheduler assigns a task to an available worker thread, it executes the transactions in the task by iterating from the lowest T_{ind} to the highest T_{ind} (block order).

The state transition of a worker thread is shown in Figure 4.5. After being initialized, a worker thread is admitted to the thread-pool that is used by the scheduler. In the

thread-pool, a worker thread is in ready state, waiting for a task to be assigned. Once a worker thread receives a task from the scheduler, it transitions to running state and starts executing the task. During execution, if the worker thread detects a conflict, it reverts the state changes performed by the transaction and creates a merge request to inform the conflict handler (see Section 4.6) of the new conflict. Afterwards, the worker thread returns to the thread-pool and transitions to the ready state. Similarly, if a worker thread sees an abort flag, it stops the execution and revert back to the ready state. Otherwise, upon completion of the task, the worker thread becomes ready again to receive a new task. Worker threads exit when the scheduler notifies them that no more tasks that belong to a block are available.

An individual transaction's status can be one of the three: running, executed or waiting. Every transaction has the waiting status before it is executed. During execution, a transaction with the running status may encounter errors according to the transaction logic, such as running out of gas or failed assert conditions. The transaction is considered failed; however, it is still a valid transaction that is marked executed and is included in the block. A transaction with running status always finishes the execution and becomes executed. A worker thread may decide to re-execute the transaction, so it discards the state changes, and sets the transaction's status back to the waiting.

Worker thread iterates through the task following the block order. If it encounters an un-executed transaction, the worker thread executes it using the latest state and keeps the state changes locally in the buffer. The worker thread may encounter an already executed transaction due to the merging of the tasks, as explained in Section 4.6, in the past. It validates the reads performed by such transactions to ensure that no stale reads have occurred. An executed transaction may incur stale reads when a prior transaction, with lower T_{ind}, gets executed or re-executed and it updates a state object that was previously not observed by the executed transaction with higher T_{ind} (more on this in Section 4.6). The executed transaction requires re-execution with updated state. A worker thread uses a task specific Watchlist to find any stale reads. The Watchlist is a collection of key-value pairs where a key is an updated state object and the corresponding value is the smallest index of the transaction that updated the state object. After detecting a stale read, the worker thread discards the state changes of the executed transaction and re-executes it. Old discarded writes along with the new updated objects are added to the watchlist to detect stale reads incurred by subsequent executed transactions in the task.

Each worker thread uses an Ethereum virtual machine(EVM) to execute a transaction. A worker thread creates a thread local instance of EVM. As EVM executes transactions, it may read/write to state objects. A write operation updates object that are buffered locally in the thread. Subsequent transactions in a task can read the updated object from

the buffer and avoid expensive database lookups. We attach a logger to the EVM to track the reads and writes of the state objects during the transaction execution. This generates a read-write set that is used to determine the transaction conflicts.

Each worker thread has a local set of owned and shared keys, where a key represents a unique ID of a state object. These keys are populated using the pre-execution read-write estimates of all the transactions in the task. The owned keys are the keys that the thread exclusively owns, allowing it to update them. While the shared keys are shared among multiple threads. The transaction's read-write keys are validated to find conflicting access after the execution. Each individual write is first locally checked within the set of the owned keys. To pass validation, each individual write must be owned by the thread. If it is not already owned, the thread requests the key manager to grant the ownership of the key. If it succeeds the key is added to the set of owned keys. Note that a worker thread only calls the key manager if a transaction accesses data outside the read-write set generated from pre-execution step, otherwise the accesses are guaranteed to be safe. Similarly, a transaction's reads pass validation if the thread either shares or owns the keys. If the validation fails, the worker thread stops execution, uses the EVM's undo log to revert the transaction's state changes to remove invalid writes, and creates a merge request with the conflicted worker thread. One worker thread can potentially conflict with multiple worker threads. However, only one conflict is resolved at a time by merge operation. Eventually, all conflicts will be resolved to successfully finish task execution. The merge request notifies the conflict handler of the new conflict. The worker thread returns to the thread-pool and the task waits for the conflict to be resolved before being re-executed. Otherwise, if the validation passes, the transaction is marked as executed.

Finally, after a transaction execution, a worker thread checks for the interrupt signal from the conflict handler. If it is set, the thread aborts and returns; otherwise, it moves to the next transaction in order.

4.5 State Storage

Ethereum's complete state size exceeds 1TB [2], and it continuously expands as blocks are added to the chain. Nodes store the full state tree in the database, while partial global state is stored in memory as shown in Figure 4.6. If the required state object is not available in memory, the object is fetched from the database on demand. During the execution, all EVM instances have a consistent view of the global state therefore they can safely read data from memory. State updates are only applied to the global state during the commit phase when all transactions are executed. The EVM instruction that initiates

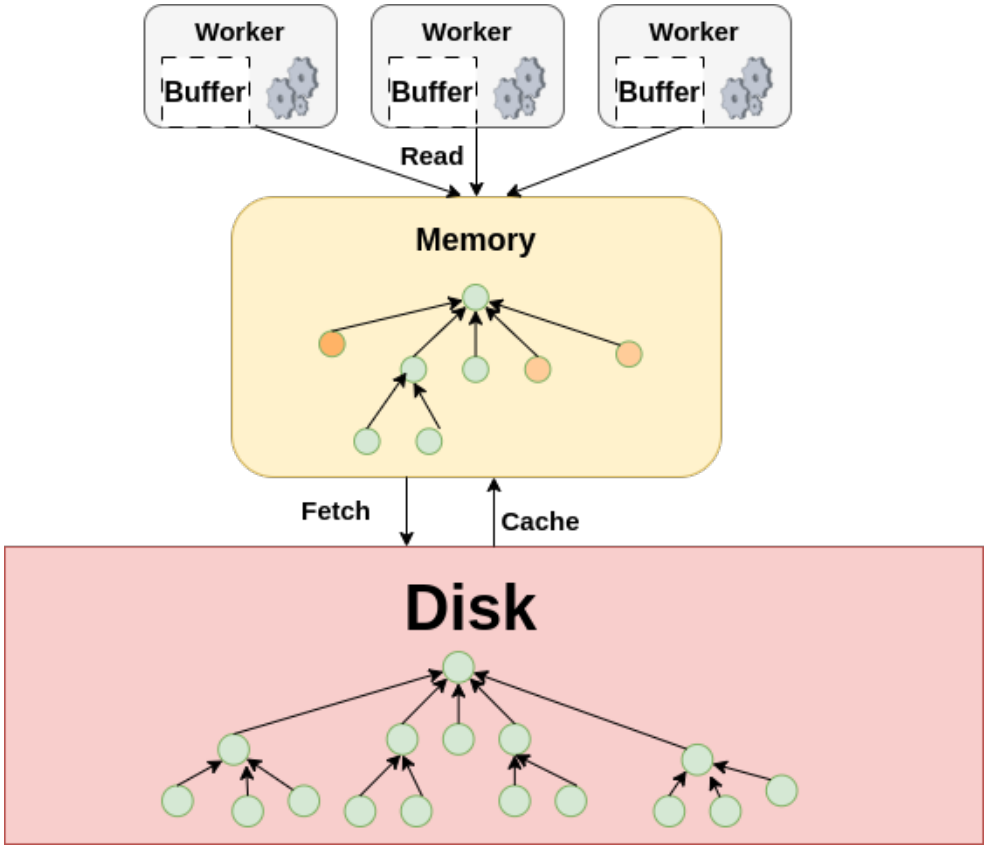

Figure 4.6: State storage in Block-X. Partial state is kept in memory, from which worker threads read data and keep it in local buffer

data lookup first checks the local buffer before checking the shared memory and eventually the database. Once the data is retrieved, it is kept in the local buffer, and subsequent read/write operations use the data in the local buffer. Once all transactions in the block are executed, the batch commit operation is called that moves modified state objects from local buffer to the memory. Data present in memory is periodically moved to the database if it exceeds a threshold size.

We modified the write operation of Go-Ethereum client[3] to support copy-on-write so that a write to a state object creates a new version instead of updating in place. The version number of a state object is assigned using the T_{ind}, referring to the transaction that updated the state. These new versions are kept in the thread's local version store. The version store keeps the state objects sorted according to the versions so that a transaction

can perform binary search to lookup a particular version of a particular state object. Most transactions use the latest version so they first check it before performing the binary search. The version store enables Block-X to: 1) read a specific version of the state object for a transaction execution and 2) revert state by discarding any state changes of the preceding transaction. Multiple versions of the data facilitate transaction conflict resolution which we will describe in Section 4.6.

Block-X maintains multiple versions of the state objects, which takes up extra space. Old versions can be garbage collected after the batch commit operation.

4.6 Conflict handler

The key manager is a part of the conflict handler that detects a conflicting access performed by a worker thread. It is a module that interfaces with the task. It keeps track of the owned and shared keys of all threads. A thread that executes a task may invoke the key manager to detect any potential conflicting data accesses. Threads only call the key manager when the accessed data does not exist in their thread-local set of owned/shared keys. Then, the accessed key must be the key that was not predicted to be accessed during the pre-execution step. During the critical path execution, state accesses may differ from the estimated accesses generated in the pre-execution step because of two main factors: 1) Unlike the pre-execution step that relies on the base state snapshot to estimate state accesses for all transactions, critical path execution strictly follows block order and uses up-to-date state. 2) Block metadata, including timestamp and hash, varies, and its value remains unknown until the block creation time. Therefore, any transaction that uses the block metadata as input may now have a different control flow due to the updated values.

If a thread performs an un-predicted write, a value that is not in task's set of owned keys, then the worker thread requests to exclusively own the key. The key manager checks that the requested key is not owned and shared by any other thread. If successful, the worker thread gains ownership of the key without encountering a conflict. Similarly, a request to share a key is handled by checking that the key is not owned by any other thread. If the request fails, it is marked as a conflict. A conflict arises because two transactions executed by two worker threads now have a new unpredicted transactional dependency. If it is a WW dependency, worker threads must stop execution and merge state because during the commit phase, the worker threads are allowed to commit state in parallel. So the worker threads must update mutually disjoint state. With RW dependency, a transaction with higher T_{ind} fails to observe writes performed by a transaction with lower T_{ind}, resulting in an abort.

When a conflicting (source) worker thread detects a conflict, it creates a merge request to notify the conflict handler to abort destination worker thread's execution if it is in the running state. This resolves conflict sooner to prevent further transaction execution from encountering stale reads. The conflict handler aborts the destination worker thread execution by setting an abort flag. Upon seeing the abort flag, the conflicted (destination) worker thread stops execution and waits for the merge operation. After seeing both threads in waiting state, the conflict handler dequeues the merge request and starts the process of conflict resolution. Although merge request are enqueued concurrently, they are dequeued and handled sequentially. Block-X introduces two techniques to resolve conflicts:

Discard-and-reexecute: A simple method that does not require copy on write of the state objects. It discards all the prior work by removing the state changes in the local buffer of both worker threads. This rolls back all of the state update performed by the worker thread because all the state changes resulting from transaction execution are kept in local buffers; they do not affect the global state until the commit step. A new dependency edge is added to the dependency graph. Afterwards, it merges the source task into the destination task by performing union operation on owned keys, shared keys and the transactions. Then, the transactions are sorted by T_{ind} to maintain the block order. The new merged task now consists of a larger connected component that comprises of a combined set of transactions from source and destination worker tasks. Finally, the task is forwarded to the scheduler for re-execution.

Merge-and-resume: The process begins by merging the tasks, similar to Discard-and-rexecute, and the updated state objects from the source thread into the destination thread. This policy requires copy-on-write that creates a new version of a state objects upon a write. Note that the locally buffered state updates are mutually disjoint and valid. This is ensured by the source worker thread, which removes all the state changes performed by the conflicting transaction before initiating a merge request to the destination thread. The merge operation produces a state that is consistent with the state produced if all the executed transactions are applied in order. Finally, the destination task is forwarded for re-execution.

After the merge, the worker thread iterates from the beginning of the merged transactions following the block order to find any transaction that remains un-executed or is executed with stale data. An un-executed transaction is executed using the latest state object versions from prior transactions (with lower T_{ind}) in order. While an executed transaction is validated to find if it performed any stale reads. This involves checking if any read object is part of the watchlist and if the watchlist object has a smaller T_{ind} than the transaction. This means the executed transaction did not observe a state update from a recent prior (with lower T_{ind}) transaction execution. The transaction is re-executed using

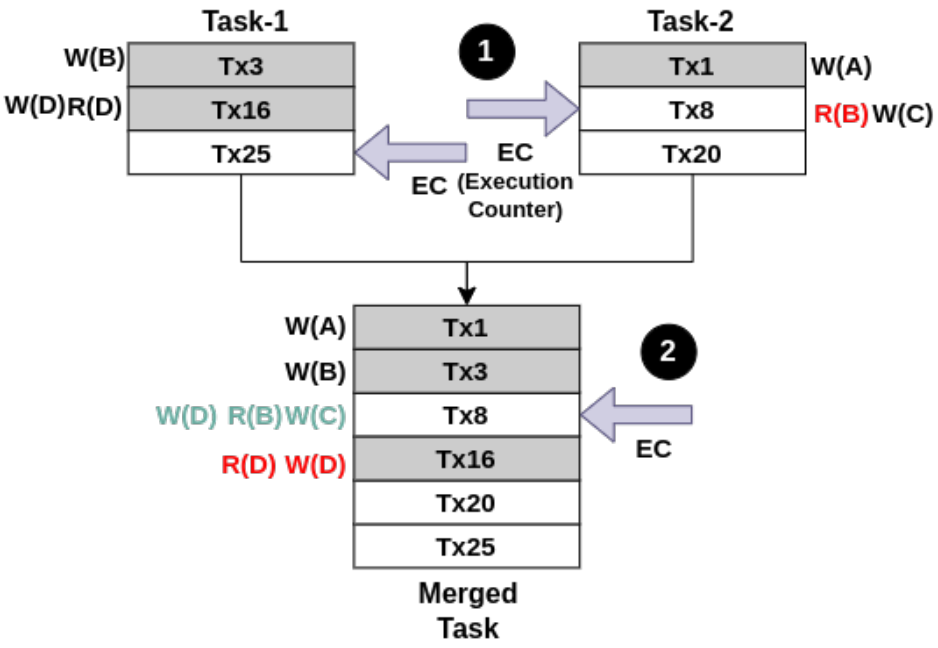

Figure 4.7: Example of Merge-and-resume. During the parallel execution of Task-1 and Task-2 in step-1, Tx8 RW conflicts with Tx3 by performing read on object B. After the merge, in step-2, worker thread iterates from the beginning of the merged task and successfully validates any stale reads performed by Tx1 and Tx3. It executes T8 using the updated version of object B from Tx3. Tx8 now additionally updates object D. Tx16 is already executed, however it will subsequently fail validation because an updated object D now exists from prior transaction (Tx8) execution.

the updated version of the state object. Generally, if a transaction is executed/re-executed in the presence of a subsequent executed transaction, as depicted in Figure 4.7, the transaction can update a state object that was not observed by previously executed transaction with higher T_{ind}. This create a stale read and a RW dependency that did not exist before. Similarly, when a version of state object is discarded to perform a transaction re-execution, reads performed by the subsequent executed transactions on the discarded data become invalid. All such updated state objects are added to the watchlist along with the T_{ind}. A worker thread determines a transaction that can invalidate subsequent transactions by simply keeping track of the highest T_{ind} of the executed transaction in a task.

An example of Merge-and-resume is illustrated in Figure 4.7. During the parallel execution of the Task-1 and Task-2, Transaction 8 (T8) reads object B that creates a RW

dependency with T3 leading to a conflict. The conflict is resolved by merging the task-1 and task-2 along with the state updates. After the merge, worker thread iterates from the beginning of the merged task and successfully validates T1 and T2 of any stale reads. Afterwards, it executes T8 using the updated version of object B from T3. T8 now additionally updates object D. All the T8 writes are included in the watchlist because it may invalidate subsequent executed transactions. Afterwards, the worker thread encounters T16, an executed transaction, and validates it using the watchlist. The worker thread find that it read object D that is now updated by T8. The validation will fail and T16 will be re-executed using the updated version of object D from T8.

The conflict handler may encounter merge requests that form a cycle. One such example would be when two tasks conflict with each other and create merge requests at the same time. It is important to note that this does not imply a cycle in the dependency graph. Although merge requests are enqueued concurrently, they are handled sequentially. After the merge operation, the source task is marked with a special flag to denote that it is completed and merged with the destination task. Subsequent merge requests to the source task are then forwarded to the destination task. This process continues until a destination task that is not marked as completed is found.

4.7 Correctness

We now argue that Block-X guarantees a more restrictive form of serializability where the result of its parallel execution schedule must be equivalent to the one in which transactions are executed serially following the block order.

Serializability The execution schedule is generated following the dependency graph as defined in Section 4.2. If there is a dependency edge $(t_j \rightarrow t_i)$ in the dependency graph, then t_j must be executed after t_i as it must observe the updates performed by t_i. Using the conflict serializability theorem [42], if a conflict graph, which represents conflicts between transactions, is acyclic, then the execution schedule is serializable. The dependency edges in the dependency graph sufficiently capture conflicts between transactions. Therefore, we use the dependency graph as a conflict graph to show that it is acyclic.

First, we assume that no new dependencies are added to the dependency graph built by the dependency builder module. By definition, given a set of transactions $T = \{t_1, ..., t_n\}$, if there is a dependency edge from t_j to t_i $(t_j \rightarrow t_i)$ in the graph, then j must be greater than i. If the graph contains a cycle, then there exists transactional dependencies: $t_k \rightarrow t_j \rightarrow ...t_i \rightarrow t_k$ such that $k > j > i$. This implies that there exists a dependency edge from $t_i \rightarrow t_k$ where $i < k$ which is not possible.

The dependency graph has no cycle. So for every dependency $t_j \rightarrow t_i$ where $j > i$, we can have a topological ordering [27] O where t_i is ordered before t_j $O(t_i) < O(t_j)$. Worker threads execute dependent transactions sequentially while maintaining this order. In other words, if there is a dependency edge $(t_j \rightarrow t_i)$, t_j is executed after t_i by observing the state updates performed by t_i. Concurrent execution only happen for transactions that do not have the dependency and can be ordered arbitrarily. Parallel execution based on this order yields the same outcome as serial execution following the topological ordering. A topological ordering of the graph that break ties using the T_{ind} will produce an order identical to the block order.

During the execution, new dependencies may arise that are resolved by merging the tasks. Merge operation adds a dependency edge by connecting two conflicting transactions. A new dependency edge to the graph cannot create a cycle because, according to the definition, a dependency edge only exists from a higher transaction index to a lower transaction index $(t_j \rightarrow t_i, j > i)$. Therefore, the execution schedule is still serializable. The new dependency may invalidate executed transactions with higher T_{ind}. This is handled using one of the two conflict handling policies: 1) Discard-and-reexecute 2) Merge-and-resume. We will argue that the final state produced after resolving a conflict using these policies is equivalent to the state produced when all transactions in the merged task are executed sequentially in block order.

Discard-and-reexecute: It completely discards prior work and trivially re-execute all the transactions sequentially in block order.

Merge-and-resume: Merge-and-resume first merges locally buffered updated state objects. The state objects belonging to the tasks prior to the merge are mutually disjoint because the conflict handler only allows parallel execution of transactions that do not conflict. Transactions in both tasks are executed sequentially in block order prior to the merge, and executed transactions in one task do not conflict with any executed transaction in the other task. So, the merged task represents a state that is equivalent to the the state produced if all the executed transactions are applied in block order. Note that some transactions may not be executed and the merged state may not be equal to the state produced if all the transactions in the merged task are executed sequentially. Transaction execution in Block-X is deterministic: the state output after executing a transaction can only change if and only if the input to the transaction changes. Note that during the critical path execution, seeds that can create randomness such as block timestamp are already determined.

In order to ensure that all transactions in the merged task are executed with the most up-to-date state in block order, the worker thread iterates from the first to the last

transaction in the task following the block order and checks the following:

a) If a transaction is un-executed, it executes the transaction using the state objects from the latest preceding dependent transaction. If no preceding transaction updated the required input state, this means the transaction has no dependency on other transactions. Therefore, it is safe to use the base state from the database because this must be the first transaction to use the state and it is not available in the local buffer of the worker thread. If any preceding executed transaction (with lower T_{ind}) updated the input state, then the updated state version exists due to the copy-on-write operation. The transaction must be able to observe the updated version because the execution follows the block order and the transaction is only executed once all the transactions it depends on, according to the dependency graph, are already executed.

b) If a transaction is already executed, it validates that all inputs/reads are the most up-to-date according to the current execution of the task. This is achieved by checking the watchlist, which contains the state objects that were updated/written whenever a transaction was executed or re-executed while a subsequent executed transaction (with higher T_{ind}) exists in the task. If no such subsequent executed transaction exists, stale/invalid reads are not possible within the task. For clarity, the scenario is also depicted in Figure 4.7. If any stale read is found, the transaction's writes are discarded, and it is re-executed using the most up-to-date state.

This ensures that all transactions in the task will finish execution in block order with the most up-to-date state. This produces the state equivalent to the state produced if all the transactions in the task are executed sequentially in block order. All the tasks that successfully finish execution do not conflict with any other task and together they contain all the block transactions. It follows that the final state produced by all the successfully executed tasks is equivalent to the state produced if all transactions in the block are executed sequentially in block order.

4.8 Limitation

Every transaction pays a gas fee to become part of the blockchain. It consists of a base fee and a priority fee known as a tip. For a transaction to be considered valid, it must meet the minimum gas fee requirement. Additionally, transactions can offer a tip to incentivize miners to prioritize their inclusion in the block. As a result, each transaction execution involves updating the miner's state object, creating a WW dependency and making the whole execution sequential.

Block-X removes this bottleneck by delaying the tip payment to the miner until all the block transactions in a block finish execution. While this approach enables parallel execution, it restricts miners from utilizing the gas tip immediately after the transaction execution. Instead, miners can only start using the amount earned from the tip in the subsequent block. Note that this is a minor change in the Ethereum protocol. Block-X is still compatible with the EVM chains.

4.9 Example contract

We now provide an example that describes a token swap service provided by a popular decentralized exchange called Uniswap-V2. It implements a factory contract and a number of distinct token pairs contracts. Uniswap operates on the principles of automated market making[8] where pairs of assets are stored in pooled reserves. It serves as a token exchange, and a very popular transaction is for a user to exchange *Token-A* for *Token-B*.

A simplified token swap logic is outlined in Algorithm 1. Consider a specific instance of a token swap, such as A to B; the swapToken function first finds the address of the pooled reserve to swap tokens. It transfers the amount to the pair (AB) (line 2) contract and then calls the swap function (line 4). Swap function is a part of the pair contract that transfers the tokens to the caller address and update the reserve values of both A and B pool. The reserve amounts are used to determine the exchange rate as per the constant product formula [8]. The transfer function updates the caller's balance in the ERC-20 contract with the updated values of both token-A and token-B.

Overall, the token swap only requires updates to the reserves in AB pair contract and to the balances in ERC-20 token contract. Other users conducting different token swaps, such as token-C to token-D, can be handled in parallel since their actions do not involve interactions with the AB pair contract and token contract. Uniswap has thousands of contract pairs. It can greatly improve throughput if trades are executed in parallel.

Algorithm 1 Uniswap Token Swap

1: **function** SWAPEXACTTOKENSFORTOKENS
2: *pair* ← pairfor(factory, token-A, token-B)
3: safeTransferFrom(sender, pair)
4: pair.swap(amountAOut, amountBOut, sender) ▷ Swap the token A→B

pair contract:

1: address factory;
2: uint112 reserveA;
3: uint112 reserveB;
4: **function** SWAP(amountAout, amountBout, to)
5: $\_transfer$(token-A, amountAOout, to)
6: $\_transfer$(token-B, amountBOut, to)
7: updateReserves(reserveA, reserveB)

ERC-20 contract:

1: mapping(address → uint) balanceOf;
2: **function** _TRANSFER(from, to, value)
3: balanceOf[from] ← balanceOf[from] - value
4: balanceOf[to] ← balanceOf[to] + value

Chapter 5

Evaluation

We compare the performance of Block-X against state-of-the-art serial and parallel execution approaches using real-world workload.

Testbed. We conduct our experiments on a cluster node having Intel XeonD D-1540 @ 2.00GHz CPU with 64GBs of RAM, 1TB of SSD storage, and 1Gbps network connection. The node has 2 sockets, each having 10 cores, with two threads sharing one core. It runs Ubuntu 20.04.3 with Linux Kernel 5.13.0. We use Ethereum Geth version v1.11.6 with default configuration for all the experiments. We use the default snap sync mode [6] to sync the node with the Ethereum Mainnet.

Workload. Our experiments use 3000 Ethereum blocks on Mainnet with block numbers from 17607300 to 17610300. These blocks have 393,183 transactions. An Ethereum block consists of simple transfer transactions and smart contract transactions. Transfer transactions are simple transactions that move funds from one account to another. They do not access keys outside their read-write sets. Smart contract transactions are more complex and one contract can invoke other contracts. Smart contracts may access keys outside their read-write sets. In our dataset, 70% of transactions are smart contracts, and 30% are simple transactions. Each block has a gas limit of 30 million gas units. Gas is the unit that measures the amount of computational effort required to execute instructions on Ethereum. The number of transactions in the block varies because different transactions require different amount of gas depending on the transaction logic.

Alternatives. We compare the throughput of the following approaches:

- *Ethereum* implements serial approach where transactions are executed one after the other in a sequence.

- *Pessimistic* approach explicitly declares read-write sets. The system uses that information to execute transactions in parallel. Once a transaction execution is complete, the system checks and rejects a transaction if it accesses a key outside the declared read-write sets. Solana and Sui chains adopt this approach. Note that this may cause subsequent dependent transactions to fail, creating cascading effect. We implement the Pessimistic approach by modifying the Go Ethereum client [3].

- *Block-X*. We implement Block-X by modifying the Go Ethereum client [3]. Unless otherwise specified, the evaluation uses Block-X with merge-and-resume policy to handle runtime conflicts.

5.1 Throughput Evaluation

We compare the throughput of the transaction processing step of the three alternatives. Figure 5.1 shows the throughput of the three systems. The results show that Block-X achieves 1.9 $\times$ higher throughput than Ethereum, and comparable performance to the pessimistic approach.

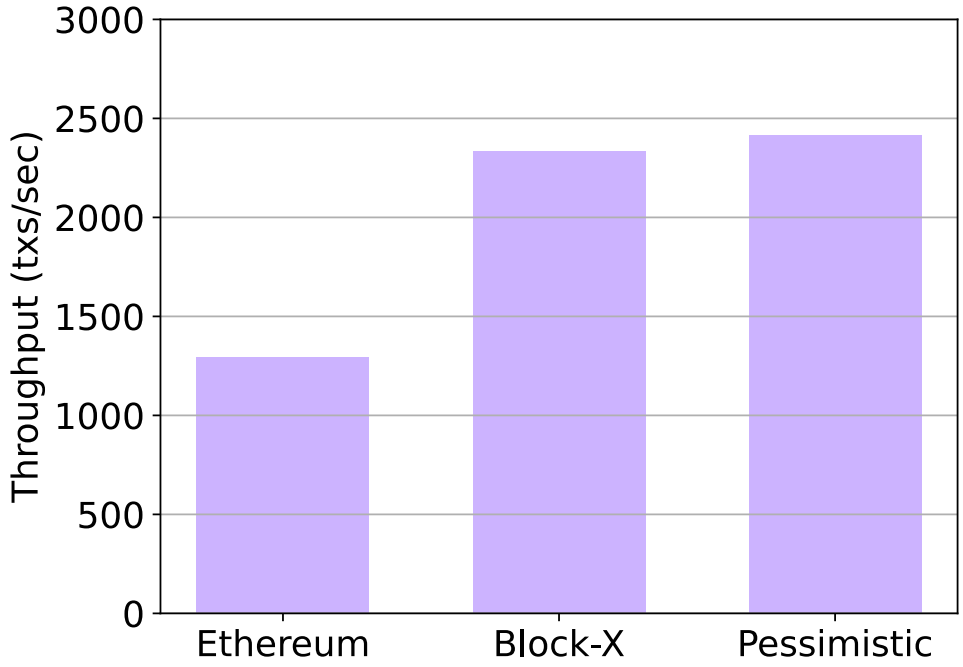

Figure 5.1: Throughput of the different systems.

To run transactions in parallel, Block-X builds a dependency graph of transactions in a block, then finds connected components in the graph. Block-X uses one thread to execute transactions in a connected component. To understand the level of parallelism present in the workload, Figure 5.2 shows the number of connected components present in a block and Figure 5.3 shows the amount of gas used by the largest connected components. The gas in the graph is an indicator of the computational effort of the connected component. While Figure 5.2 shows that 97% of blocks have 4 or more connected components, Figure 5.3 shows their computational overhead is highly imbalanced with the largest component representing 37% of the workload's computational effort. This indicates that the parallel execution is bottlenecked by the serial execution of transaction in the largest connected component. With the largest connected component, the maximum theoretical speedup possible is 2.7×. Block-X achieves 1.9× speedup. It falls short of the theoretical limit due to overhead incurred from conflict resolution, storage, and scheduling.

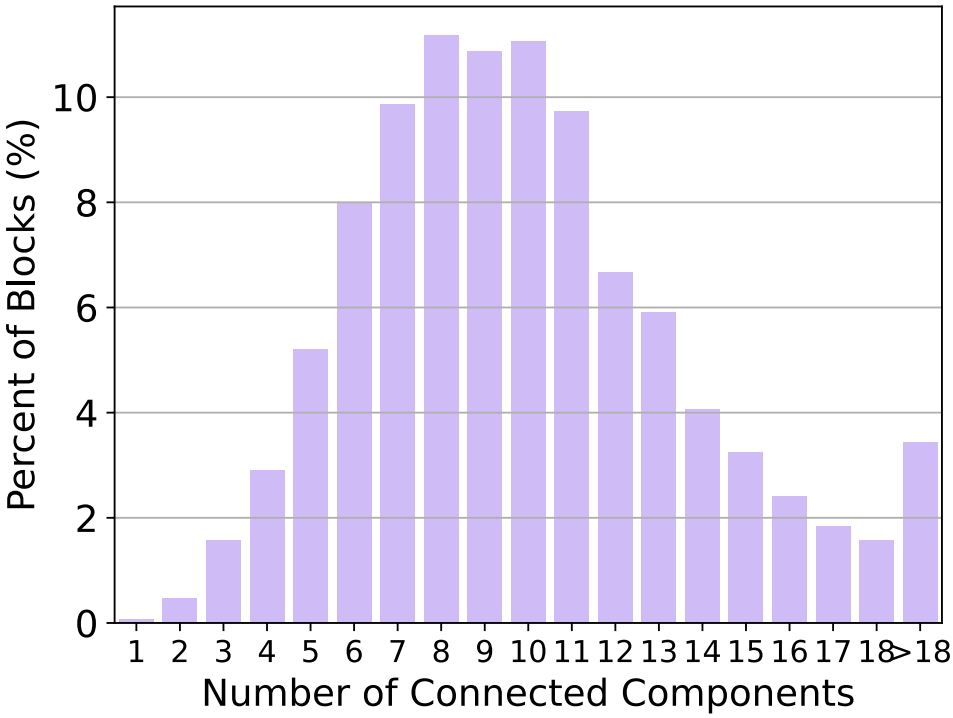

Figure 5.2: Percentage of blocks with given number of connected components.

We study the viability of executing transactions in a single connected component in parallel. Our results in Section 5.1 show that Block-X throughput is limited because of the serial execution of the largest connected component (Figure 5.3). We analyze the transactions in the largest connected components in all blocks and found that there is limited opportunity to execute transactions in parallel. We sort the connected components by size and found the longest path in the components using depth first search algorithm. Figure 5.5 shows the size of largest connected components, with the largest one containing 24% of the block transactions while independent transactions consists of 53%. The size of the longest path within these connected components is shown in Figure 5.5. This shows that the longest paths in the connected components consist of atleast (87%) of the component transactions, forming a chain of dependencies. This indicates limited opportunities to parallelize the execution of a single connected component. Overall, this shows that a small number of contracts are popular among users.

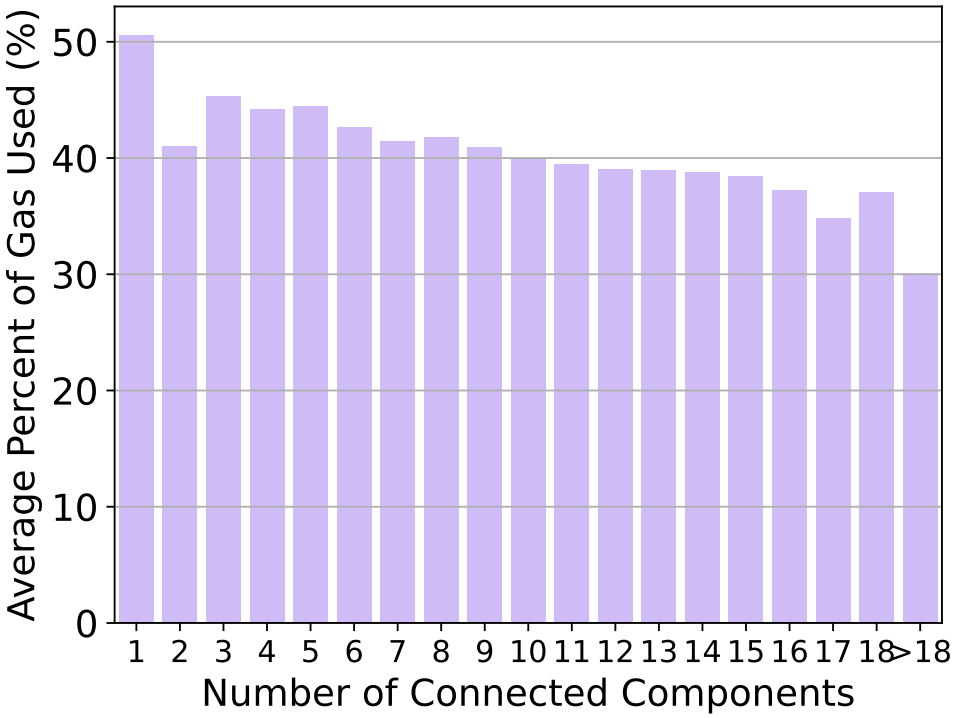

Figure 5.3: Average of gas used by the largest connected components in a block

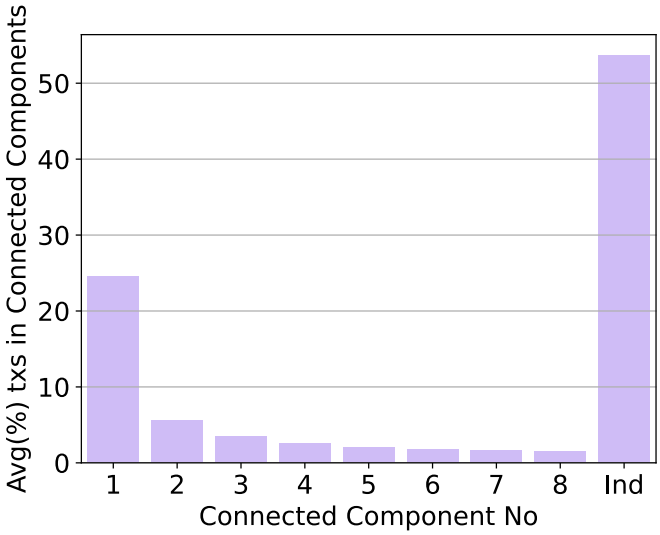

Figure 5.4: Percentage of block transactions in largest connected components. Ind denotes independent transactions

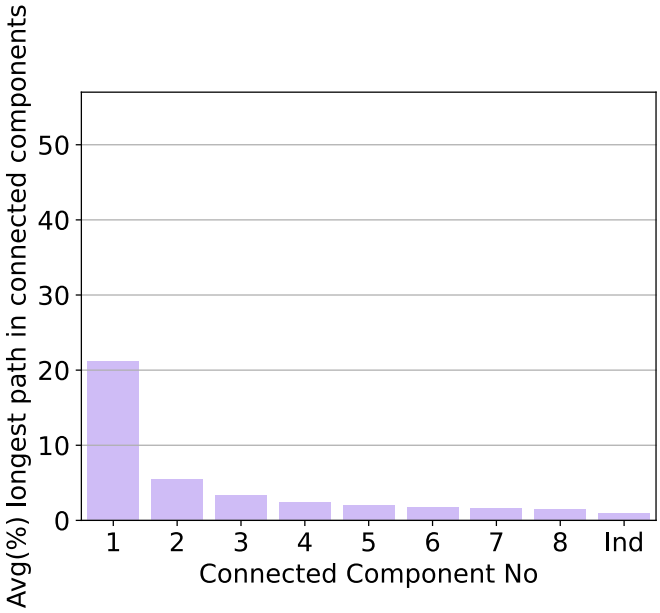

Figure 5.5: Percentage of block transactions in the longest path of the largest connected components. Ind denotes independent transactions.

Block-X achieves comparable performance to the pessimistic approach without rejecting any transactions. The pessimistic approach doesn't have a conflict resolution mechanism. It simply detects access violations and rejects transactions after a transaction completes execution. This wastes system resources by discarding executed transactions that can be valid. In our workload, the pessimistic approach rejects 4% of transactions in total. We also plotted the number of rejected transactions against the percentage of the blocks in Figure 5.6. It shows that 97% of the blocks have at least 1 transaction that accesses data outside the estimated read-write set. In Block-X, less than 1% of transactions result in conflict, triggering the conflict resolution mechanism.

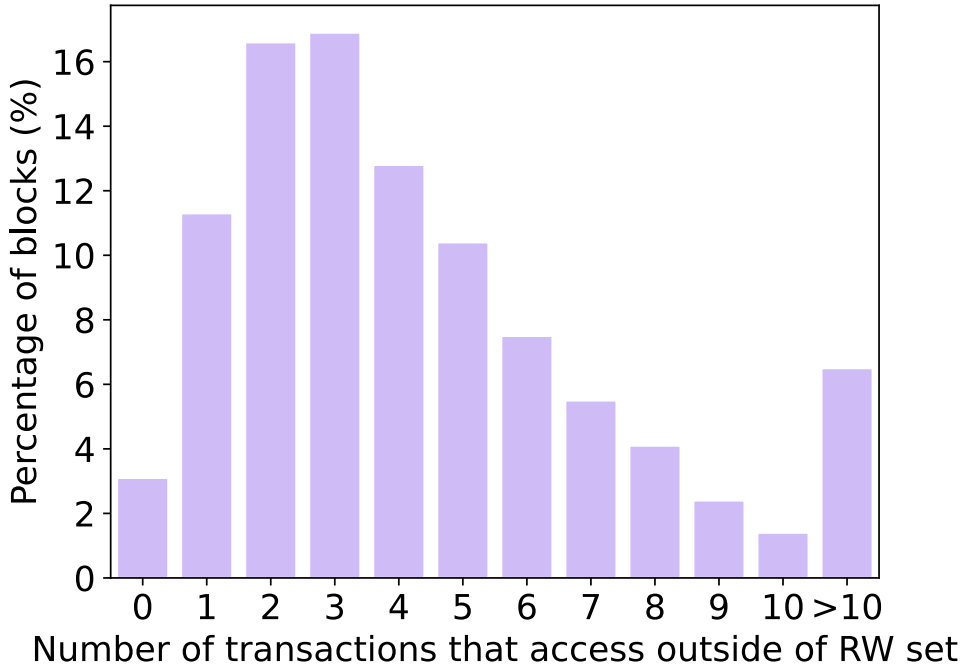

Figure 5.6: Percentage of blocks with number of transactions accessing data outside of their read-write set

5.2 Block Execution Time

We evaluate the time it takes to complete the execution of all transactions in a block. Figure 5.7 shows the CDF of the block execution time of the three systems.

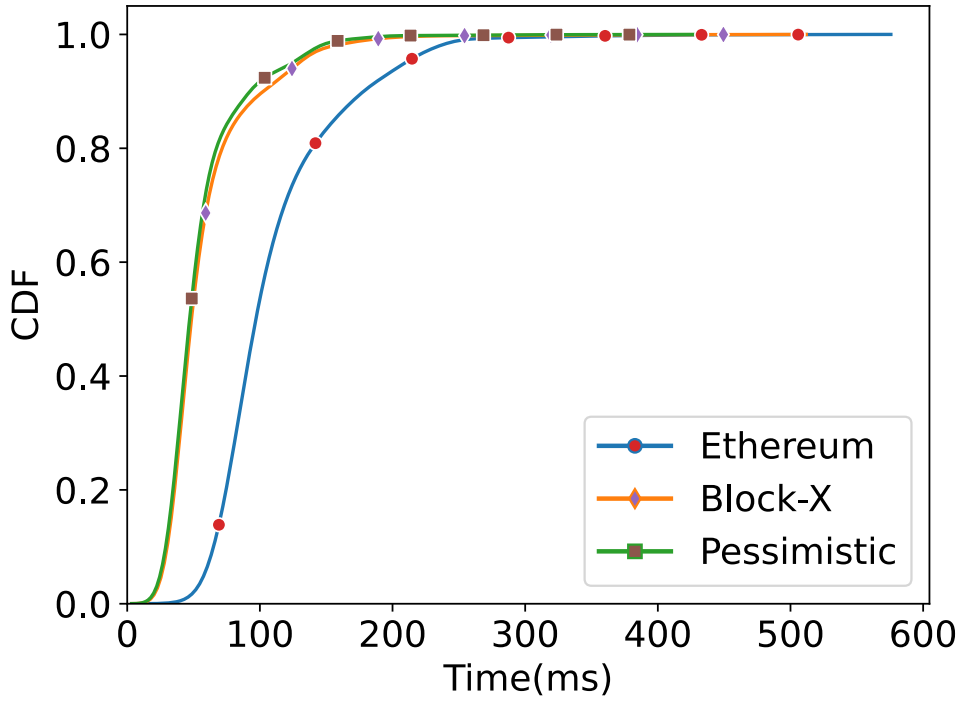

Figure 5.7: Block execution time.

Figure 5.7 shows that Block-X noticeably improves the block execution time compared to Ethereum by parallelizing the block execution. Block-X reduces the median block execution time by 50% and the 95 percentile by 37.5%. Figure 5.7 shows that Block-X has a shorter latency tail. Block-X achieves a comparable performance to the pessimistic approach without rejecting any transactions. In our workload, the pessimistic approach rejects 4% of transactions because they access data outside the read-write sets.

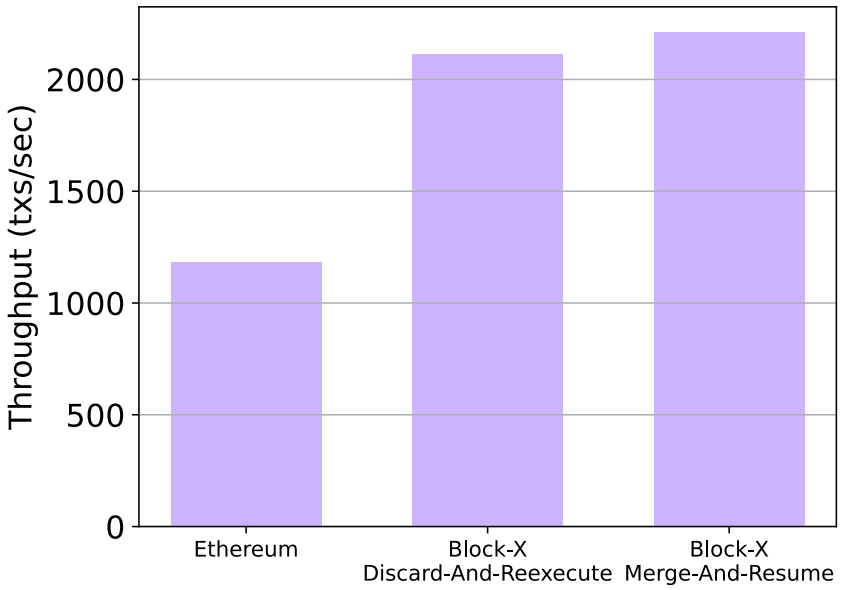

Figure 5.8: Throughput of Block-X's conflict resolution policies.

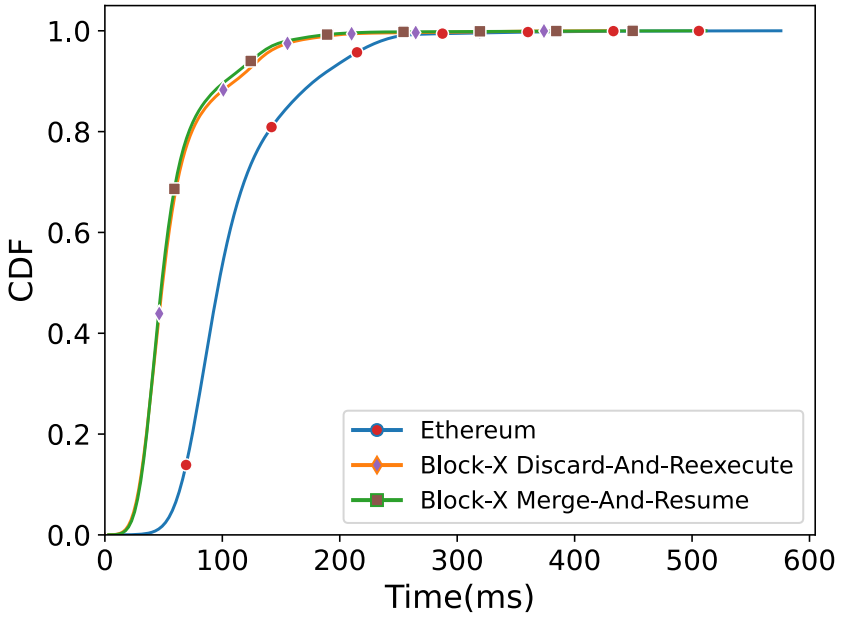

Figure 5.9: CDF of execution time with different conflict handling policies

5.3 Performance of Conflict Resolution Mechanism

We compare the performance of the two conflict resolution policies: Discard-and-reexecute and Merge-and-resume. Figure 5.8 shows the system throughput, and Figure 5.9 shows the block execution time. Our results show that the both techniques have comparable performance. Discard-and-reexecute may unnecessarily repeat work when conflict arises. Figure 5.10 shows the number of conflicts in blocks. The figure shows that 70% of blocks do not have conflicts and over 90% of blocks have one conflict or none. Merge and resume use copy-on-write to handle conflicts without discarding prior work. Merge and resume does less work when it resolves conflicts.

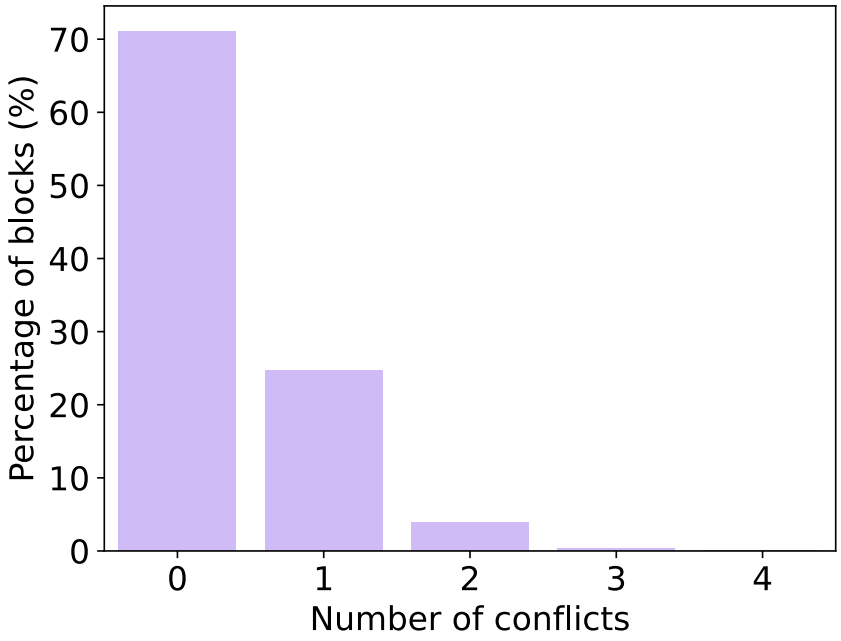

Figure 5.10: Number of conflicts leading to merge in parallel execution

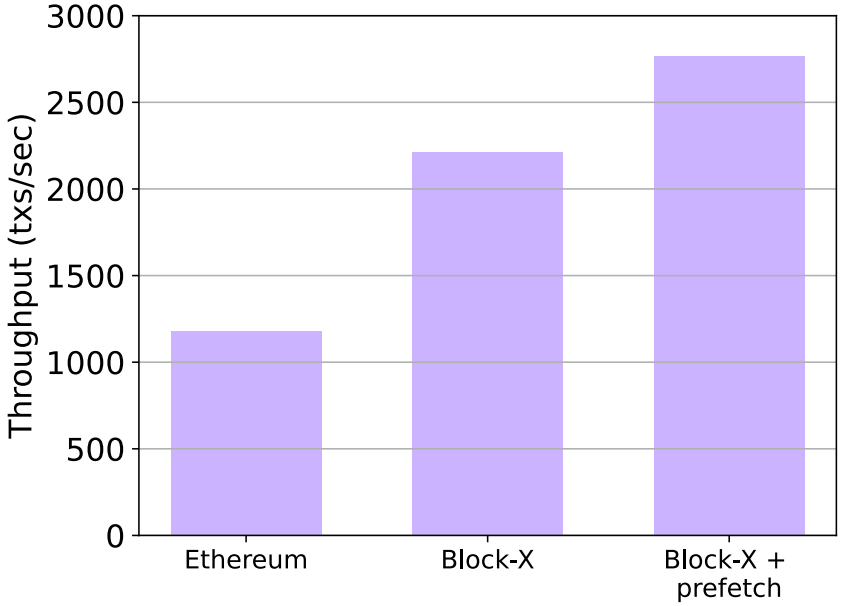

Figure 5.11: Throughput with prefetching.

5.4 Pre-fetching

We evaluate the benefits of data prefetching. We cache the data accessed during the pre-execution step in memory to improve the performance of the transaction execution on the critical path. This reduces the disk accesses during the critical path execution. Figure 5.11 shows the system throughput with prefetching. It shows that prefetching improves Block-X throughput by 24%. Block-X with prefetching achieves 2.34× higher throughput compared to Ethereum.

5.5 Validator Evaluation

To accelerate validating a block, the validator in Block-X executes transactions in parallel using the execution schedule created by the miner. Validators are responsible to check that the state transition due to the transaction execution is valid. A miner can add an invalid schedule that may give different output due to nondeterministic interleaving of the data accesses. Validators don't trust miners, therefore, they independently validate the block

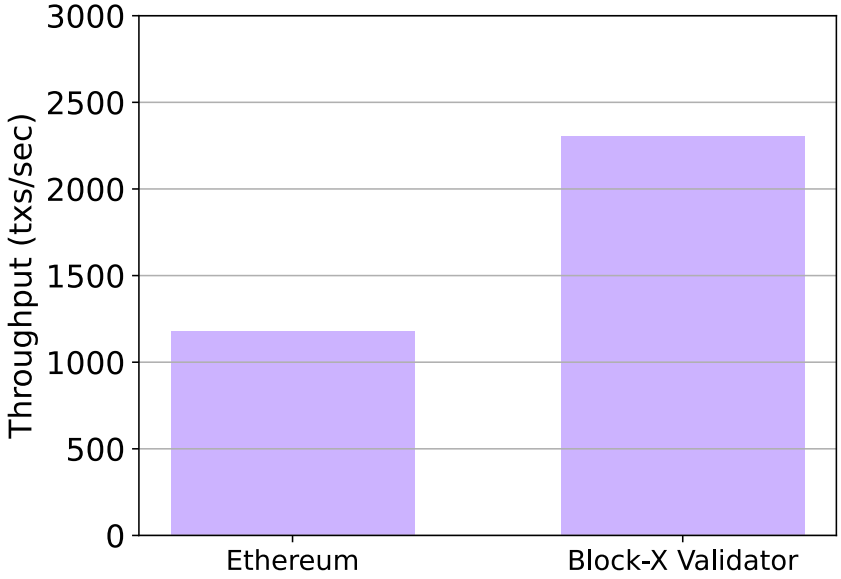

Figure 5.12: Block-X validator throughput as compared to the Block-X validator

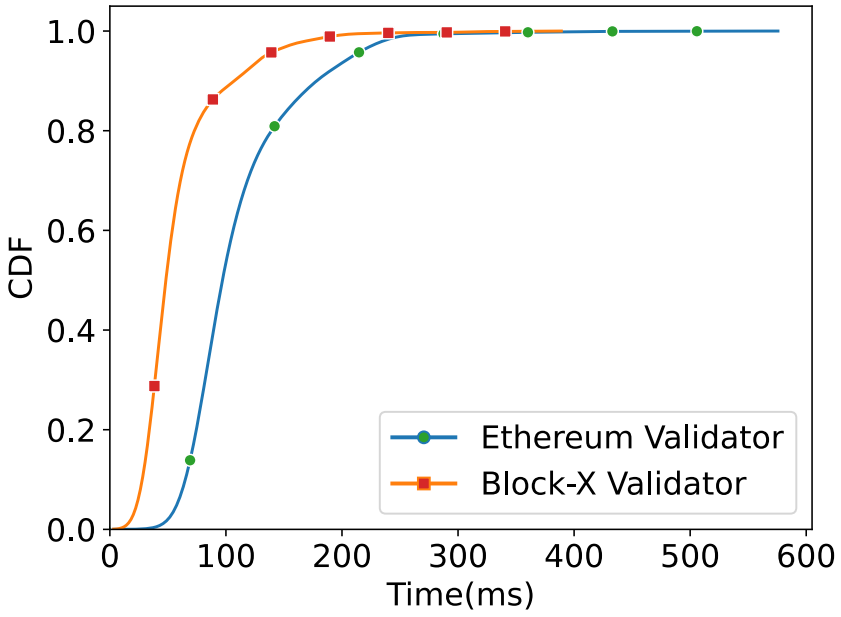

Figure 5.13: Block-X validation time.

by re-executing all the transactions. If a validator detects conflicts between transactions, it rejects the block. Figure 5.12 compares the throughput of Block-X validator to Etheruem validator and Figure 5.13 shows the block execution time. Block-X validator achieves 1.95 $\times$ higher validation throughput leading to 50% lower median block execution time.

Chapter 6

Conclusion

We propose Block-X, a parallel transaction execution system for public blockchain systems. On the Ethereum mainnet, it achieves up to a 2.3× improvement in block execution time. Block-X optimistically executes transactions while they are in the mempool to determine the read-write keys. Then, it uses this information to build a parallel execution schedule. This approach minimizes conflicts on the critical execution path. Block-X is compatible with Ethereum. The final state produced after the parallel execution matches the state generated after the sequential execution in block order.

For future work, we plan to explore two directions. First, we intend to explore a technique that combines static and dynamic analysis of transactions to estimate the read-write sets. Second, we aim to design and investigate a mechanism for miners to construct blocks that facilitate highly parallelizable transaction execution. Additionally, building decentralized applications that avoid using the same state objects can enable more opportunities for parallel transaction execution.

References

[1] Ethereum average gas limit chart. https://etherscan.io/chart/gaslimit. [Accessed 28-03-2024].

[2] Ethereum full node state with default settings. https://etherscan.io/chartsync/chaindefault. [Accessed 29-03-2024].

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

[4] Merkle patricia trie. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/. [Accessed 29-03-2024].

[5] Parallel Execution — docs.monad.xyz. https://docs.monad.xyz/technical-discussion/execution/parallel-execution. [Accessed 23-01-2024].

[6] Sync modes. https://geth.ethereum.org/docs/fundamentals/sync-modes. [Accessed 28-03-2024].

[7] Cryptokitties craze slows down transactions on ethereum, Dec 2017.

[8] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020.

[9] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.*, 2021.

[10] Jameela Al-Jaroodi and Nader Mohamed. Blockchain in industries: A survey. *IEEE access*, 7:36500–36515, 2019.

[11] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1337–1347. IEEE, 2019.

[12] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *2016 2nd international conference on open and big data (OBD)*, pages 25–30. IEEE, 2016.

[13] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

[14] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[15] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper. *arXiv preprint arXiv:2003.03052*, 2020.

[16] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 570–587, 2021.

[17] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE access*, 4:2292–2303, 2016.

[18] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains: (a position paper). In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[19] 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*, pages 303–312, 2017.

[20] 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.

[21] Sui Foundation. All about parallelization, Jan 2024.

[22] Hector Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[23] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

[24] 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*, pages 51–68, 2017.

[25] Anton Hasselgren, Katina Kralevska, Danilo Gligoroski, Sindre A Pedersen, and Arild Faxvaag. Blockchain in healthcare and health sciences—a scoping review. *International Journal of Medical Informatics*, 134:104040, 2020.

[26] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[27] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[28] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[29] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[30] Sei Labs. Sei: The layer 1 for trading. https://github.com/sei-protocol/sei-chain/blob/main/whitepaper/Sei_Whitepaper.pdf. [Accessed 17-03-2024].

[31] D. Z. Morris. If bitcoin can't handle a few jpegs, how can it handle the world? https://www.coindesk.com/consensus-magazine/2023/05/10/if-bitcoin-cant-handle-a-few-jpegs-how-can-it-handle-the-world/, May 10 2023. Accessed: 19 August 2023.

[32] Richa Naidu. Nestle, Unilever, Tyson and others team with IBM on blockchain. https://www.reuters.com/article/us-ibm-retailers-blockchain/nestle-unilever-tyson-and-others-team-with-ibm-on-blockchain-idUSKCN1B21B1/. [Accessed 17-03-2024].

[33] Richa Naidu. Nestle, unilever, tyson and others team with ibm on blockchain. https://www.reuters.com/article/us-ibm-retailers-blockchain/nestle-unilever-tyson-and-others-team-with-ibm-on-blockchain-idUSKCN1B21B1/, August 22 2017. Accessed: 17 March 2024.

[34] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[35] Gareth W Peters, Efstathios Panayi, and Ariane Chapelle. Trends in crypto-currencies and blockchain technologies: A monetary theory and regulation perspective. *arXiv preprint arXiv:1508.04364*, 2015.

[36] Maciel M Queiroz, Renato Telles, and Silvia H Bonilla. Blockchain and supply chain management integration: a systematic review of the literature. *Supply chain management: An international journal*, 25(2):241–254, 2020.

[37] Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. *Available [online].[Accessed: 4-12-2018]*, 2018.

[38] Sei. Sei v2 - the first parallelized evm blockchain. https://blog.sei.io/sei-v2-the-first-parallelized-evm/. [Accessed 18-03-2024].

[39] Guy L Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231, 1989.

[40] The MystenLabs Team. The sui smart contracts platform. https://docs.sui.io/paper/sui.pdf.

[41] VISA. Visa fact sheet. https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf. [Accessed 20-03-2024].

[42] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

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

[44] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

[45] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.