Optimizing Smart Contract Parallelism via Commutative Operations

Sidnei David Martins Teixeira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Paolo Romano
Prof. Miguel Ângelo Marques de Matos

Examination Committee

October 2024

Acknowledgments

First, I would like to express my sincere appreciation for the dedication and perseverance I brought to this work. This thesis is a reflection of my commitment to consistency and integrity in pursuit of the person I strive to become. By staying true to myself and to my habits, I have ensured that every decision made during the development of this thesis aligned with the principles I hold. It is through this ongoing effort that I can confidently say I am moving toward the person I aspire to be.

I would also like to sincerely thank my dissertation advisors, Prof. Miguel Matos and Prof. Paolo Romano, for their guidance, time, and openness throughout this journey. Their willingness to discuss any challenges I faced, as well as their insightful suggestions and encouragement to approach problems thoughtfully, has been far more valuable than simply being handed solutions. Their mentorship has deeply enriched this experience.

A special thanks to Francisco Rola for his time, availability, and the knowledge he generously shared, which also contributed significantly to the success of this thesis.

Lastly, I am immensely grateful to all my friends who have supported me throughout my academic journey. A heartfelt thanks to Afonso Pinto and Ricardo Rocha, whose constant encouragement has pushed me to grow both intellectually and personally, fostering a more open-minded and tolerant approach to challenges.

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT), using national funds as part of the projects INESC-ID UIDB/50021/2020, , Ainur (financed by the OE with ref. PTDC/CCI-COM/4485/2021) and Composable's bilateral project ParallelCosmwasm.

Abstract

Blockchain technology has emerged as a revolutionary paradigm, enabling decentralized and secure transactions across various domains. Originating with Bitcoin as a pioneer cryptocurrency system, and later with the appearance of more versatile systems like Ethereum, which introduced Smart Contracts, blockchains now host several secure decentralized applications (dApps) governed by immutable contracts deployed in the ledger. However, these systems fail to exploit the inherent concurrency of today's multicore architectures by adhering to the conventional approach of executing transactions sequentially. As consensus protocol developments introduce new and more efficient algorithms, the new bottleneck is shifting towards transaction execution speed. Several works have been developed with the aim of solving this limitation, but they either rely on methods that compromise the security of the blockchain by appending dependency-tracking graphs in the produced block, lack transparency by imposing alterations to the system, or do not take into consideration the hot-spot problem - most transactions access only a few storage locations. We propose a strategy that employs static analysis to detect conflicting operations as well as operation commutativity to parallelize operations aimed at the same contract keys. Our results show the proposed solution achieved a speedup of $8.57\times$ for workloads without commutativity support, and up to $10.1\times$ speedup for commutative intensive host-spot workloads.

Keywords

Blockchain; Smart Contracts; Concurrency; Symbolic Execution;

Resumo

A tecnologia blockchain emergiu como um paradigma revolucionário, permitindo transações descentralizadas e seguras em vários domínios. Originando-se com o Bitcoin como o sistema pioneiro de criptomoedas, e mais tarde com o aparecimento de sistemas mais versáteis como o Ethereum, que introduziu os smart contracts, as blockchains agora hospedam várias aplicações descentralizadas seguras (dApps), governadas por contratos imutáveis implantados no registo da blockchain. No entanto, estes sistemas não exploram a concorrência inerente às arquiteturas multicore modernas, ao aderirem à abordagem convencional de execução sequencial das transações. À medida que os desenvolvimentos nos protocolos de consenso introduzem algoritmos novos e mais eficientes, o novo gargalo tende a ser a velocidade de execução das transações. Vários trabalhos têm sido desenvolvidos com o objetivo de resolver esta limitação, mas ou dependem de métodos que comprometem a segurança da blockchain ao anexar grafos de dependências ao bloco produzido, ou carecem de transparência ao impor alterações ao sistema, ou não levam em consideração o problema dos *hot-spots* — a maioria das transações acede apenas a alguns locais de armazenamento de um número reduzido de smart contracts. Propomos uma estratégia que utiliza análise estática para detetar operações em conflito bem como a deteção de operações comutativas de modo a paralelizar operações referentes ao mesmo item. Os nossos resultados mostram que a solução proposta atingiu um speedup máximo de $8.67\times$ para padrões de acesso sem suporte para comutatividade, e atingiu um speedup de $10.1\times$ para padrões de acesso de hot-spots rico em operações comutativas.

Palavras Chave

Blockchain; Smart Contracts; Concorrência; Execução Simbólica;

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

ABCI	Application Blockchain Interface
BFS	Breadth-First Search
CVM	CosmWasm Virtual Machine
DAG	Directed Acyclic Graph
DFS	Depth-First Search
EOA	Externally Owned Account
EVM	Ethereum Virtual Machine
HTM	Hardware Transactional Memory
IBC	Inter-Blockchain Communication Protocol
OCC	Optimistic Concurrency Control
OCC-DA	Optimistic Concurrency Control with Deterministic Aborts
PBFT	Practical Byzantine Fault Tolerance
PoS	Proof of Stake
PoW	Proof of Work
SDK	Software Development Kit
STM	Software Transactional Memory
TM	Transactional Memory
WASM	WebAssembly
WVM	Wasmer Virtual Machine

1 Introduction

As the interest in secure and decentralized applications intensifies, blockchain has emerged as a central foundation for diverse applications across domains in recent years. Essentially, blockchains converge Distributed Systems and Databases, featuring a distributed data structure for recording transactions. This structure is maintained by nodes in a peer-to-peer paradigm, devoid of a central authority.

Although several popular systems have been developed in recent years, blockchain technology trace back to 1991, with Haber *et. al* [4], where the first idea of a chain of signed blocks was originated. The primary idea was to guarantee no alterations of data items are possible without the system noticing, by implementing a chain of cryptographically signed items whose signature depends on all previous items. The concept evolved over the years, defining an immutable data item as a block of transactions. It gained global attention in 2008 with Bitcoin [5], introduced by Satoshi Nakamoto, marking the launch of the first decentralized cryptocurrency. Later, with the introduction of runnable code deployed on the blockchain, commonly known as Smart Contracts, Ethereum [6] elevated the versatility of blockchain systems. Not only they allow for simple transfers across user accounts, but also enable the development of complex logic containing conditional statements that serve as immutable and automated decision making machines running on a decentralized environment.

In the general workflow of blockchain systems, *clients* submit transactions which are broadcasted to all nodes participating in the system in a peer-to-peer fashion. These transactions, ranging from simple transfers to invocations of deployed contracts, may alter the blockchain's state. When a node accumulates a sufficient number of transactions, it takes on the role of proposing and executing the transactions to form a new block. To maintain consistency among all participating nodes, blockchain systems typically combine a consensus algorithm to agree on the order of transactions and State Machine Replication (SMR) [7] to ensure that all nodes apply the agreed-upon transactions in the same order, resulting in consistent state across the network.

Following the proposal of a block by a node, other nodes, acting as validators, verify the correctness of each transaction within the proposed block. Their goal is to ensure that the results align with those of the proposing node. If the validators confirm that the transactions and their effects match the expected outcomes, the block is accepted. Else, the block is rejected, and all associated changes are discarded.

Blockchain systems fall into two main categories: **Permissioned** systems involve identified nodes with limited trust, using asynchronous fault-tolerant protocols like PBFT [8]. In contrast, **Permissionless** systems have no assumptions about node identities, allowing free entry and exit. Blockchains use less strict consensus algorithms, offering weaker consistency guarantees - Eventual Consistency [9]. While two nodes in permissionless systems may temporarily append different blocks, they are assured to eventually converge on the exact same state. Bitcoin [5], Ethereum [6] and Cosmos [10] are well-known permissionless systems.

1.1 Motivation

The rising popularity of smart contracts is attracting numerous applications to leverage blockchain characteristics such as auditability, persistency, and decentralization [11]. This surge not only leads to a growth in Decentralized Applications (dApps) but also drives an increase in their complexity and performance requirements. Despite being one of the most popular blockchain platforms, Ethereum [6] currently achieves an average of only 15 transactions per second (TPS) [12].

As new advances in consensus algorithms have been proposed within academia [11], yielding more efficient solutions for achieving the desired consistency, the bottleneck is shifting from consensus to transaction execution and validation stages where nodes need to execute every transaction within the block. While the traditional approach of executing transactions sequentially ensures determinism, it falls short of fully leveraging the capabilities of current multi-core architectures present in system nodes.

This current limitation presents an open opportunity to explore concurrency in transaction execution. However, the concurrent execution of smart contract transactions is not straightforward as different transactions can perform conflicting accesses to shared data. Two transactions conflict if they access the same object, with at least one performing a write operation. Simply executing these transactions arbitrarily may lead to data races, resulting in an inconsistent system state. For systems such as Bitcoin [5], shared data is limited to the **sender** and **receiver** accounts, whose address is statically predeclared in the transaction, easily allowing static determination of the transaction's Read-Write-Set (RWS) and defining a concurrent schedule. Contrastingly, in blockchains allowing smart contract deployment, determining the RWS statically is more challenging, given that contracts use Turing-complete languages. The **a priori** analysis may be incomplete due to **path explosion** [13], and it must also handle RWS that depend on state.

Although some works [2, 14–16] avoid this problem by using speculative execution and determining the RWS at runtime using instrumentation, achieving equivalence to the same serial execution across all nodes is challenging. Serializability, a common correctness criterion, ensures only equivalence to some

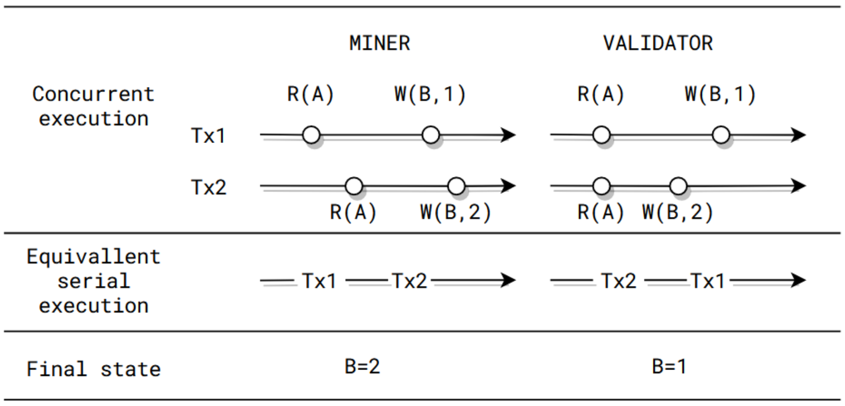


Figure 1.1: Different serial equivalence.

serial execution, allowing miners and validators to have different equivalent serial executions, leading to distinct states—a fundamental correctness and security concern. Figure 1.1 illustrates this problem, where both a miner and validator have the exact same pair of transactions to execute (`Tx1 and Tx2`), but the miner first writes the value from `Tx1` and only then the value from `Tx2`, while the validator does the inverse. Each node has a serial equivalence, although different. The miner's equivallent serial execution is `Tx1` $\rightarrow$ `Tx2`, resulting in B's value to be 2. The validator's equivallent serial execution is `Tx2` $\rightarrow$ `Tx1`, resulting in B's value to be 1, which is different from the miner's value.

In addition, nodes may behave maliciously, altering any data. Some works [15, 16] propose strategies that allow for validators to follow the miner execution schedule deterministically, at the cost of security, as malicious miners may intentionally propose altered schedules that are deliberately slow, but still ensure serial equivalence.

Furthermore, recent works [1, 2, 17] emphasize the **hot-spot problem** in blockchain systems. This issue arises when a small number of highly popular contracts are invoked by many transactions, resulting in high contention on a small set of data items. Notably, this pattern often involves commutative operations, primarily due to the nature of increments on global counters for balances or prices. Such a pattern poses challenges for classical Optimistic Concurrency Control (OCC) proposals [14, 15] due to a high number of aborts. Thus, more fine grained strategies are needed to minimize the cost of aborts under such scenarios. Furthermore, current OCC strategies do not take into consideration resource waste resulting from roll-backs and re-execution, resulting in potentially high number of re-executions of transactions with only the last execution producing effective work.

1.2 Objectives

The primary objective of this Thesis is to propose a transparent strategy for efficiently handling hot-spot scenarios by leveraging operation commutativity and employing parallel execution of such operations.

This strategy is grounded in findings from previous works [1,2,17], which highlight that hot-spot workloads frequently involve a small subset of items being altered and also invole commutative operations, such as increments to shared counters. By identifying and exploiting commutative operations, the proposed approach aims to parallelize their execution, thereby enhancing overall throughput. Specifically, this work seeks to demonstrate that, by parallelizing commutative operations, significant performance improvements can be achieved over baseline strategies that lack commutativity support.

1.3 Outline

The remainder of this document is structured as follows. Chapter 2 introduces some background concepts. Chapter 3 discusses current state-of-the-art solutions, and their limitations. Chapter 4 provides a detailed account of the proposed solution, offering a comprehensive breakdown of each component. Chapter 5 specifies the methodology for evaluating the system, as well as the criteria and metrics employed to measure its effectiveness and performance. Finally Chapter 6 concludes the document, specifying current limitations, and insights for future work.

2 Background

In this Chapter we introduce some background concepts necessary to understand the rest of the document. We start by describing the blockchain architecture and workflow in Section 2.1, taking emphasis on the Ethereum blockchain, followed by a description of Cosmos. Then, in Section 2.2 we define Transaction Memory as it has been used as a common solution on several proposed works. Finally, Section 2.3 briefly discusses Symbolic Execution.

2.1 Blockchain and Smart Contracts

A blockchain fundamentally operates as a decentralized state machine, processing a predetermined set of state-transition requests in the form of transactions, storing them in a chain of cryptographically secured blocks that acts as a tamper-proof, append-only sequence of all verified and accepted state transitions.

2.1.1 Ethereum

As a state machine, blockchains must have a well defined way of representing its state (implicitly or explicitly). Ethereum defines its state - knwon as **world state** - explicitly as a map of addresses to accounts [18]. This map is replicated and stored locally on each node as a database, containing the transactions and system state in a serialized hashed data structure called a Merkle Patricia Trie [19]. This specialized structure allows fast state equality comparison between two states by simply comparing the root hash of the trie (which by definition captures all the underlying state changes).

Ethereum adopts an account model in its state, serving as state enclosures that keep a domain logically bounded, i.e., each account contains several state attributes. Each account's state, e.g., its balance, is susceptible to changes upon transaction execution, leading to a new state. Ethereum defines two different types of accounts - **Externally Owned Account (EOA)**, and **contract accounts**. EOAs have no code associated with them and have a cryptographic pair of keys which grant the possessor of the private key control over the account's funds. On the other hand, contract accounts do not have a private key, as they are owned by the logic of its smart contract code - a compiled program code recorded on the ledger, and executed by Ethereum's state machine - Ethereum Virtual Machine (EVM).

In general, blockchain systems allow clients to have accounts associated to a unique cryptographic key pair composed of a private and a public key. A private key guarantees ownership over the account, as it is used to digitally sign transactions. As such, it is never transmitted or stored on the system, and should remain private and never appear in any message passed over the network. Public keys on the other hand are publicly available, and are used to uniquely identify each account as well as to validate the signatures of transactions.

In terms of state, an account comprises of the following four fields [18], which capture details at a particular world state:

- **nonce:** Number of transactions sent from this address, or number of contract creations made by this account.
- **balance:** Monetary amount owned by this address expressed in Wei.
- **storageRoot:** 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account.
- **codeHash:** Hash of the EVM code of this account's contract.

EVM and Smart Contracts

Upon receiving enough transactions from clients, miners batch a defined number of transactions, and execute those transactions sequentially. All Ethereum nodes run the EVM - a stack-based deterministic execution environment - for all state-changing operations when executing transactions. Each transaction is decomposed into EVM bytecode instructions that are executed sequentially, altering state and recording the new state in a sandbox environment. If the execution finishes successfully, the nonce of the sender is incremented by one and the new state is written to the ledger, ensuring persistency. Note that nonces are essential to avoid replay attacks. If the execution encounters any problem, the EVM ensures all changes are rolled-back and never persisted in state. Ethereum defines three primary types of transactions: **value transfer**, which comprises a monetary transfer of **ether** from the sender to the recipient, **contract creation**, used to deploy a new smart contract, and **contract call**, which invokes the execution of compiled code associated with the recipient account.

Simply put, a Smart Contract is a Turing-complete program written in a high-level programming language (such as Solidity [20]), compiled and stored in the blockchain under a specific **contract account**. Clients can then invoke any public method of any available contract given they have the necessary funds.

The deployment of smart contracts involves encoding the contract into EVM bytecode and sending it as a transaction to the network. Miners include this transaction in a block and execute the `CREATE` operation on the EVM, which creates a new contract account. The resulting transaction receipt contains the address of the newly deployed contract.

As part of their complex logic, contracts can invoke other contracts through **message calls**, enabling autonomous information exchange. This requires distinguishing between the **sender** (`msg.sender`) and the **transaction originator** (`tx.origin`). The sender represents the address initiating the current contract call, dynamically adapting to the immediate account (either an Externally Owned Account or a contract account) invoking the function. In contrast, the transaction originator refers to the original Externally Owned Account that initiated the transaction, remaining constant regardless of the number of contract invocations in the transaction stack.

Each smart contract has its own state, and this state is isolated from the state of other contracts. This means that a contract cannot change the state of another contract directly. A contract can only change the state of another contract via a message call, making the target contract change its own state. Smart contracts, just as any account, also have a balance. Smart contracts can receive ether through direct transfers from EOAs, as well as receiving ether as part of transactions or contract creations. The balance is then used to cover any costs relative to **message calls** the contract may invoke as well as the execution cost itself.

Transactions

For state to evolve over time, there must exist state transitions. Transactions serve as the embodiment of such transition requests allowing Ethereum accounts to interact with each other. A transaction is a cryptographic set of signed instructions issued by a Client and consistently includes two fields - **sender** and **receiver**, both identified by their respective public key. Each transaction is executed by the state machine, recording temporary state changes, which are made persistent upon a successful execution, marking a new state recorded on the ledger.

Gas and Payment

To handle Turing-completeness and prevent smart contracts from running indefinitely, the EVM assigns a cost to each bytecode operation, expressed as **gas**, and evaluated in *ether*. These operations encompass a wide range, including arithmetic, control-flow, and crucially, those interacting with memory or state—such as `MLOAD`, `MSTORE*`, `SLOAD`, `SSTORE`. `MLOAD` loads a word from memory, `MSTORE*` saves a word or a byte to memory, `SLOAD` loads a word from storage, and `SSTORE` saves a word to storage.

When a transaction triggers the execution of a smart contract, its sender must define an amount of gas - **gas limit** - that sets the upper limit of what can be consumed running the contract. The gas limit set is, in turn, restricted by the sender's account balance. If the cumulative execution cost surpasses the specified gas limit, the computation is abruptly terminated, and any changes made to the state during the execution are rolled back. Consequently, a client cannot issue an execution with a gas limit higher than its own balance and anticipate a successful transaction execution. This guarantees there can never be an infinite execution that halts the entire system. Importantly, the sender does not receive a refund for the unused gas. In addition to out-of-gas exceptions, executions may also be aborted if transactions fail to meet **require** statements, essential for ensuring state consistency. For instance, a simple transfer of a certain amount X from account A to account B necessitates the verification that A holds at least X ether. If a transaction does not satisfy such require statements, it is aborted, and the execution cost is refunded to the sender—whether its an EOA or a contract account.

As part of the broader incentive mechanism, miners play a pivotal role. Miners are rewarded not only through block rewards but also by collecting transaction fees. The gas used in a transaction is multiplied by the gas price, which is determined by the transaction sender, resulting in the transaction fee. This fee serves as an incentive for miners to include a transaction in a block, as they prioritize transactions with higher fees due to the potential economic benefit. In essence, users compete to have their transactions processed promptly by offering higher gas prices.

Consensus Protocol

After the execution step, a **miner** broadcasts a block to be agreed by all nodes using the consensus protocol mechanism. In permissionless blockchains like Bitcoin, the Proof of Work (PoW) consensus strategy is used, where nodes compete to solve cryptographic puzzles, with the first successful node broadcasting the block for validation. In Proof of Stake (PoS), miners prove ownership of currency, and methods like randomization determine the stakeholder to forge the next block. PoS is more energy-efficient than PoW, but security concerns may arise due to lower mining costs.

Figure 2.1 illustrates the generic permissionless workflow, spanning from transaction creation and proposal to execution and final validation. A client initiates a transaction, which is sent to all peers through the client's Local Parity Node. Upon reception, nodes add the transaction to the memory pool. Miners, prioritizing transactions with higher fees, construct a block that is subsequently executed sequentially by the EVM. If the execution is successful, the new block is appended to the ledger and broadcasted to all nodes for validation. Validators replay the transactions from the received block, and upon successful validation, the block is added to the ledger, and the associated transactions are removed from the transaction pool.

In permissioned blockchains, algorithms that require knowing the number of participating nodes such as Practical Byzantine Fault Tolerance (PBFT) are used. PBFT tolerates byzantine faults, allowing it to handle up to $1/3$ malicious replicas.

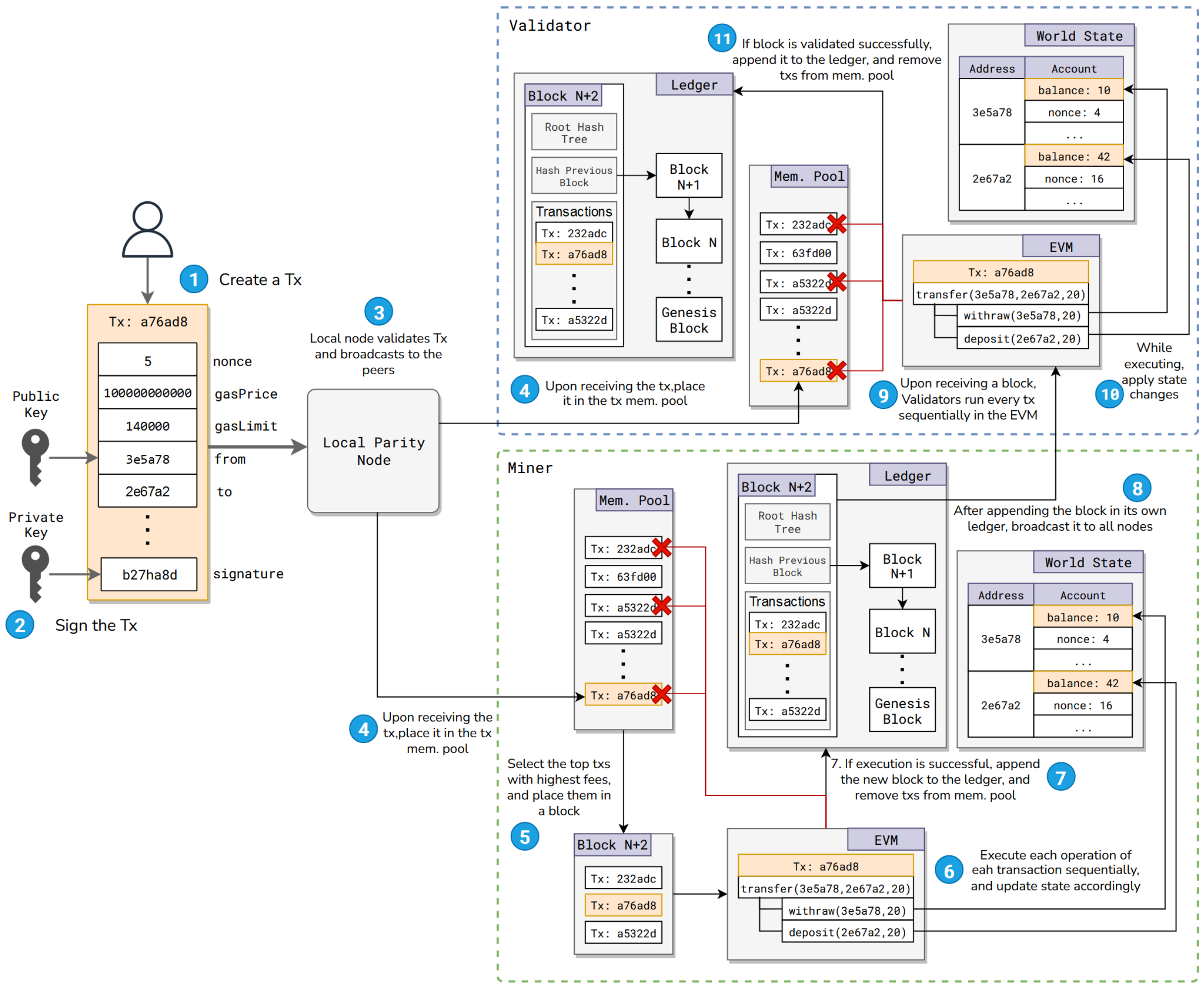


Figure 2.1: Blockchain Workflow.

The proposed work is designed to be transparent, adaptable to any blockchain type, and compatible with various consensus algorithms.

2.1.2 Cosmos

While Ethereum revolutionized blockchain with smart contracts, Cosmos [10] offers a different approach, shifting from Ethereum's monolithic architecture to a scalable model called "The Interchain"—a network of interoperable, sovereign blockchains, called zones.

Cosmos envisions an internet of blockchains, where multiple parallel chains can interoperate while maintaining their individual security properties. This approach diverges significantly from Ethereum's concept of a "world computer" where all decentralized applications share a single blockchain infrastructure. Cosmos focuses on three principles [10]:

1. **Sovereignty:** Each blockchain maintains autonomy over its governance and economic models,

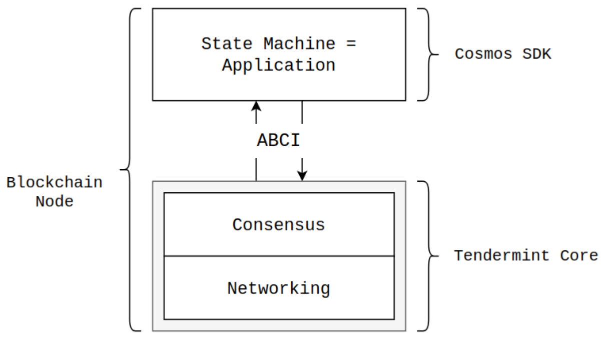


Figure 2.2: Cosmos Node.

enabling specialization for specific use cases. This contrasts sharply with Ethereum's model, where all applications follow the same EVM rules.

2. **Scalability:** Cosmos' multi-chain architecture distributes computation across blockchains, avoiding bottlenecks common in single-chain systems like Ethereum.
3. **Interoperability:** The Inter-Blockchain Communication Protocol (IBC) [21] enables seamless data and value transfer across blockchains.

Despite these differences, Cosmos retains core blockchain properties such as the account model, gas mechanism, and smart contracts, detailed in the following sections.

Blockhain Architecture

Cosmos nodes consist of three components, as shown in Figure 2.2.
CommetBFT At the heart of each independent blockchain is the Tendermint Core [22] - a PBFT-like consensus engine ensuring agreement on the blockchain's state even with faulty nodes. Tendermint is application-agnostic, enabling any deterministic application to run on the blockchain.

ABCI CometBFT (Cosmos' Tendermint Core) relays transactions to the application layer via the Application Blockchain Interface (ABCI), which the application must implement, as shown in Figure 2.2. CometBFT orders transaction bytes but does not interpret them. These ordered bytes are passed to the application through ABCI.

Cosmos SDK Virtual-machine blockchains like Ethereum introduced the concept of programmability through the use of virtual machines to interpret Turing-complete smart contracts. These blockchains provided significant advancements over prior, more rigid platforms, such as Bitcoin. However, virtual machines, like the EVM, present some limitations for more complex applications [23]:

- Smart contracts written for virtual machines are **constrained** by the capabilities of the underlying VM. For instance, the EVM requires developers to use a specific, often **immature**, programming language and provides **limited** automation and cryptographic operations.
- All smart contracts on a virtual-machine blockchain share the same resources, which can create bottlenecks and limit **performance**. Even with optimizations like sharding, the interpretation of smart contracts by the virtual machine (VM) introduces inherent inefficiencies.
- Application **sovereignty** is reduced, as decentralized applications must operate within the governance framework of the general-purpose virtual-machine blockchain. Developers have limited control over their application and are constrained by the overarching rules of the network.

To address these limitations, **Application-Specific Blockchains** have emerged, offering a flexible and efficient solution. The Cosmos SDK is designed around this concept, allowing developers to create blockchain applications that are tailored specifically to their needs,

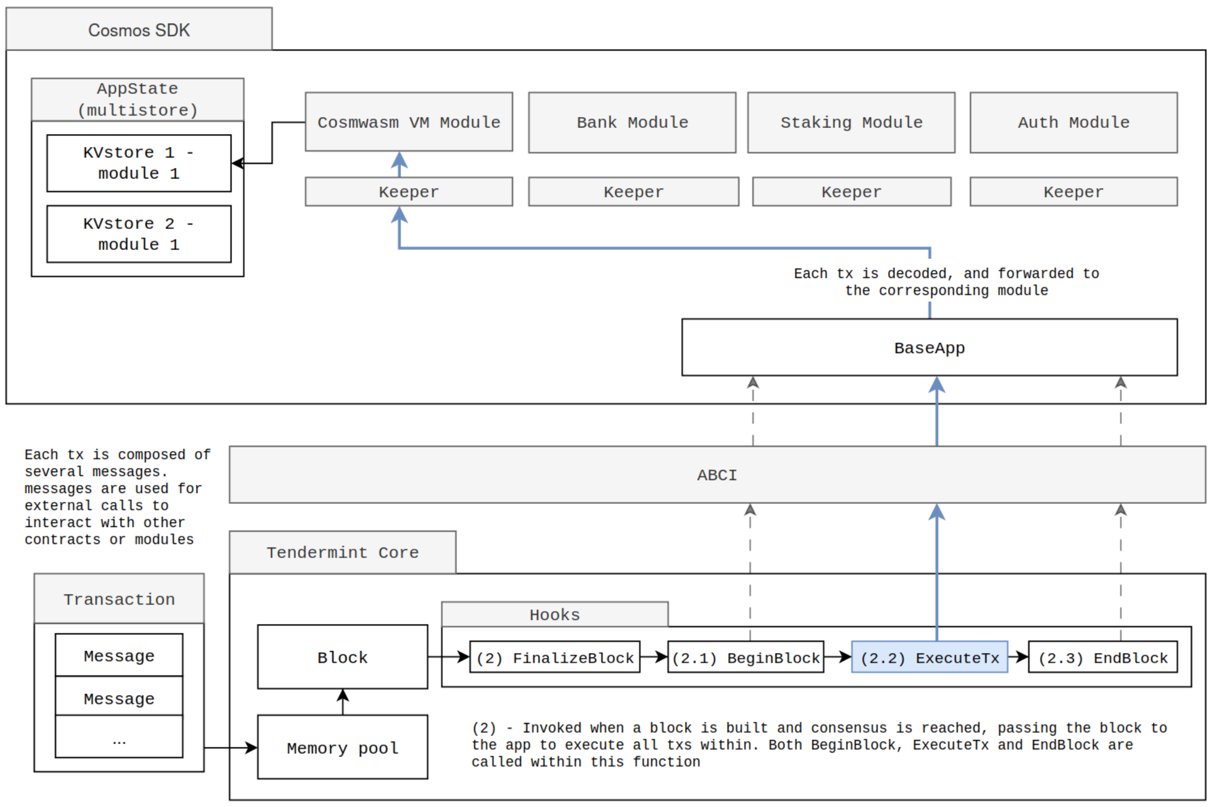
At the top level is the Application, built using the Cosmos Software Development Kit (SDK), a framework that facilitates the development of secure state-machines on top of CometBFT via a modularized framework of interoperable modules. Cosmos SDK employs a **multistore** for state management, with each module operating in its own key-value store (KVStore). Since KVStores persist only byte arrays, data must be serialized with a codec like Protocol Buffers. This design allows modules to manage their state independently, ensuring clear compartmentalization and minimal coupling.

The SDK provides several foundational modules that cover essential blockchain functionalities [24]:

- **x/auth:** Manages accounts and authentication processes.
- **x/bank:** Supports token creation and transfers.
- **x/staking:** Facilitates Proof-of-Stake consensus mechanisms, including staking operations.
- **x/slashing:** Imposes penalties on validators for misbehavior.

These core modules can be used directly or extended with additional functionality. Custom modules can also be built to define specific application logic. Modules within the Cosmos SDK interact through a well-defined security model based on object capabilities. Each module exposes access to its internal state via a **keeper**, which encapsulates the module's state and defines methods for reading and writing data. These keepers are passed between modules to authorize state access, ensuring that no unauthorized operations occur. This mechanism guarantees modular security without the need for complex access control lists.

Cosmos SDK applications are extensions of `BaseApp`, which provides core logic for transaction routing and message processing. A representation of the overall workflow is depicted in Figure 2.3. When a



Cosmos SDK
AppState (multistore)
KVstore 1 - module 1
KVstore 2 - module 1
Cosmwasm VM Module
Bank Module
Staking Module
Auth Module
Keeper
Keeper
Keeper
Keeper
Each tx is decoded, and forwarded to the corresponding module
BaseApp
Each tx is composed of several messages. messages are used for external calls to interact with other contracts or modules
ABCI
Tendermint Core
Transaction
Message
Message
...
Hooks
Block
Memory pool
(2) FinalizeBlock
(2.1) BeginBlock
(2.2) ExecuteTx
(2.3) EndBlock
(2) - Invoked when a block is built and consensus is reached, passing the block to the app to execute all txs within. Both BeginBlock, ExecuteTx and EndBlock are called within this function

Figure 2.3: Cosmos execution workflow.

transaction is received, `BaseApp` directs it to the appropriate module based on the transaction type. Each module in the Cosmos SDK integrates lifecycle hooks, which execute at various stages of the block lifecycle. This allows modules to define logic to be executed at the start of a block and at the end of a block.

IBC The IBC enables secure and permissionless data and asset transfers between blockchains, facilitating cross-chain communication and interoperability. IBC is consensus-agnostic, functioning across various networks and operating in two layers: the transport layer for secure connections and authenticated data transmission, and the application layer for processing and interpreting data packets.

IBC meets the crucial need for communication among independent, application-specific blockchains. In the Cosmos ecosystem, it standardizes ordered, exactly-once delivery of packets for exchanging state data, tokens, or assets, allowing interoperability without intermediaries. For instance, IBC allows a Cosmos blockchain to accept tokens or transfer state information securely. Modules across blockchains communicate via **channels** defined under a **connection**, with each connection supporting multiple channels.

To facilitate cross-chain communication, IBC relies on relayers that monitor the source blockchain, package relevant data, and transmit it to the target chain. Each blockchain maintains a light client of the other chain to verify incoming messages' validity without needing the entire history. The protocol supports multiple relayers for redundancy, ensuring secure message transmission across the network.

CosmWasm

CosmWasm is a WebAssembly (WASM)-based smart contract platform integrated with the Cosmos SDK, allowing the addition of smart contract functionality to application-specific blockchains. It utilizes a Wasmer Virtual Machine (WVM) for executing Turing-complete programs, typically written in Rust, in a sandboxed environment that ensures smart contracts run independently of the core blockchain application.

With CosmWasm, decentralized logic can be incorporated into applications without altering the underlying blockchain infrastructure. The CosmWasm module, `x\wasm`, serves as an intermediary that integrates smart contract execution into the modular Cosmos architecture.

Adhering to the object-capability model of the Cosmos SDK, CosmWasm contracts use keepers to interact with other SDK modules (e.g., `x\bank`) in a controlled manner. This ensures that contracts can access functionalities without directly modifying the state of other modules, preserving security and modular integrity. Figure 2.3 illustrates the workflow from block creation to byte data interpretation via ABCI, leading to the execution in the CosmWasm Virtual Machine (CVM) module.

The `x\wasm` module maintains a key-value store (KVStore) for persistent storage of smart contract bytecode and state, essential for managing operational data across blocks.

CosmWasm is also fully compatible with the IBC protocol, enabling smart contracts to communicate with other blockchains in the Cosmos ecosystem, thereby enhancing the flexibility of Cosmos-based decentralized applications.

CosmWasm Execution Workflow As discussed earlier, Cosmos blockchains utilize CosmWasm as the execution platform for smart contracts. Unlike a single, monolithic execution environment, the CVM acts as a wrapper around the WVM. It intercepts smart contract calls at designated entry points and delegates the execution of contract logic to the WVM.

The CVM manages crucial external operations such as storage access, cryptography, and other functionalities needed by smart contracts. When a contract call is initiated, the WVM executes the business logic, while the CVM handles interactions with the blockchain's state (i.e., reads and writes from storage) through external calls defined in the smart contract's WASM bytecode. This design allows the WVM to concentrate solely on smart contract execution, with the CVM synchronizing data and managing interactions with the underlying blockchain infrastructure. From the CVM developer's perspective, the smart contract operations act as a black box, with synchronization points between the smart contract logic and the CVM occurring solely through these delegations.

This architecture limits execution control, rendering strategies such as those proposed by Lin et al. [2], which rely on stack-level operation control, impractical. Such fine-grained control is abstracted away from

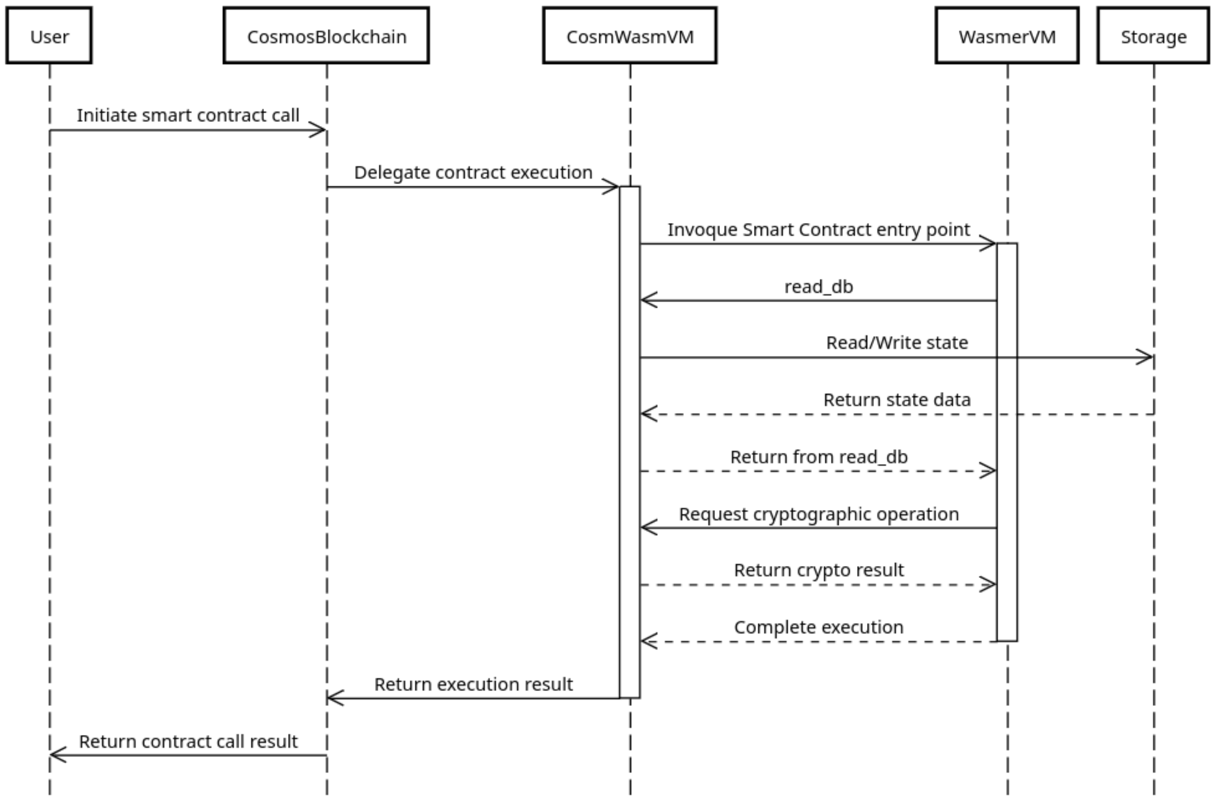


Figure 2.4: Smart Contract delegated execution.

the CVM and only available within the WVM. In response, we propose a system, as outlined in Chapter 4, that achieves operation-level control by synchronizing execution at the points where reads and writes are delegated from the WVM. Figure 2.4 illustrates the communication flow between the CVM and the WVM for a basic workload.

2.2 Transactional Memory

Transactional Memory (TM) serves as a concurrency control mechanism simplifying concurrent programming by enabling groups of load and store instructions to execute atomically, diverging from traditional lock-based synchronization methods. TM facilitates communication between concurrent threads within an application through the execution of lightweight, in-memory **transactions**. Similar to its database equivalent, a transaction represents a finite sequence of instructions that must observe certain properties such as **Serializability** and **Atomicity**. Serializability ensures that instructions from different transactions are executed without interleaving, while Atomicity mandates that a transaction either commits all changes or aborts them. During a transaction, shared data is accessed, and the transaction subsequently either commits or aborts. In the event of a conflict, one of the conflicting transactions is compelled to abort and roll back. Otherwise, it commits, and its operations are applied to the shared state atomically. The primary goal is to provide programmers with an abstraction, namely the transaction, making concurrency easier. TM proves beneficial in addressing common issues encountered in lock-based approaches, such as priority inversion, deadlocks, and reduced concurrency [25]. Transactions are speculatively executed across different threads and processors, yet only one transaction can commit in a single step. Transactional Memory can be implemented in hardware, software, or a combination of both.

Hardware Transactional Memory

Hardware Transactional Memory (HTM) operates by integrating the transaction construct and concurrency control directly into the processor unit. Within HTM systems, it monitors how memory is accessed at the physical level and manages conflicts using hardware cache hierarchies and coherence protocols. Notably, critical sections in HTM can run concurrently unless the system identifies the need for serialization to ensure smooth execution. The specific nature of HTM introduces extra rules about how transactions might fail, classifying these systems as **”best-effort”** approaches.

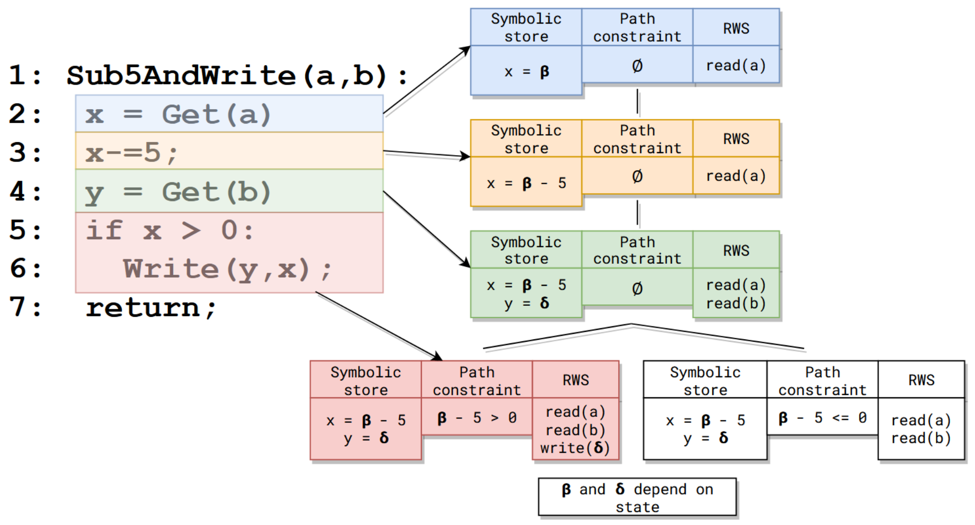
Software Transactional Memory

Software Transactional Memory (STM) systems furnish Transactional Memory capabilities through software runtime support, employing instrumentation code and software data structures. STM frameworks are compatible with prevailing hardware and offer greater flexibility compared to HTM. They serve as a more portable solution than HTM, albeit incurring overhead due to the tracking of a transaction’s memory access, which can slow down each thread up to 40% [26].

2.3 Symbolic Execution

Symbolic execution is a technique aimed to facilitate the analysis of programs by determining the classes of inputs that trigger specific execution paths. Unlike concrete execution, where a program runs on actual inputs, symbolic execution employs symbolic values for inputs, generating expressions and constraints that describe program behaviour. The exploration of multiple paths under different symbolic inputs enables a comprehensive analysis of a program’s potential outcomes.

Within symbolic execution, an interpreter traverses a program, maintaining a **first-order Boolean formula** for each explored control flow path and a **symbolic memory store** mapping variables to symbolic expressions or values, as shown in Figure 2.5. Constraints derived from these symbolic representations are subsequently solved to pinpoint inputs triggering particular branches. This approach



```
Sub5AndWrite(a,b):
 x = Get(a)
 x-=5;
 y = Get(b)
 if x > 0:
   Write(y,x);
 return;
```

Figure 2.5: Symbolic Execution.

enables reasoning about classes of inputs rather than individual values, thereby overcoming limitations linked to random concrete input generation used in unit testing.

Despite its theoretical soundness and completeness in identifying the classes of inputs for each possible output, exhaustive symbolic execution faces scalability challenges in real-world applications [13]:

- **State Space Explosion:** Language constructs like loops can exponentially increase execution states, making exhaustive exploration infeasible.
- **Constraint Solving:** Efficient constraint solving, particularly in the presence of constructs like non-linear arithmetic, is crucial. Satisfiability Modulo Theories (SMT) solvers demonstrate scalability but face challenges in certain scenarios.

Due to the high practical cost of exhaustively enumerating all program paths, SE engines adopt heuristic strategies for path prioritization, such as Depth-First Search (DFS), Breadth-First Search (BFS), or Random Path Selection. While these strategies contribute to the formulation of a more efficient search approach, the Path Explosion problem persists due to the exponential growth of states. Two common strategies employed include pruning unrealizable paths by employing constraint solvers at each branch to eliminate paths whose conditions are guaranteed to never be true, and **state merging**, a technique that combines different paths arriving at the same symbolic store. The converged path is then represented by a disjunction of the formulas describing individual states [13].

3 Related Work

In this chapter we provide an exploration on current state-of-the-art solutions. Emphasis will be placed on identifying inherent limitations, or unique contributions associated with each approach. Finally, we provide a final discussion.

3.1 Speculative Execution

Various different techniques have been proposed to improve smart contract concurrency over the years, particularly through the utilization of Speculative Execution. Works that follow this strategy do not rely on any known-in-advance information, and execute transactions in a best-effort way, hoping for the best case of reduced conflicts. Upon detecting a conflict the transaction is either delayed, or aborted, rolled-back and re-executed.

Dickerson et al. [16] introduced a strategy based on Two Phase Locking, where miners acquire abstract locks when executing transactions concurrently. If a lock acquisition fails, a conflict is detected, and the transaction is rolled back using an inverse log to undo operations. Miners then mark dependencies as a **happens-before** Directed Acyclic Graph (DAG) and include it with the block sent to validators, as shown in Figure 3.1. Validators follow a fork-join schedule, executing transactions according to the DAG order. Since the miner's strategy is non-deterministic, Dickerson et al. use the execution schedule of the block proposer as the implicit agreement, which validators follow deterministically.

This approach has some drawbacks. First, adding dependencies to the block raises security concerns, as malicious miners may attempt to propose an equivalent sequential schedule, intentionally slower than its parallel counterpart. This tactic aims to impede validation by competing miners. Although the authors suggest incentivizing miners to propose highly parallel schedules, this solution would require systemic changes, reducing transparency. Second, appending the DAG to the block increases its size, especially in workloads with many conflicts, such as **hot-spot accesses** [1, 2, 17], leading to longer transmission times and higher latency.

Saraph and Herlihy [17] proposed a simple bin-based speculative approach with two phases. First,

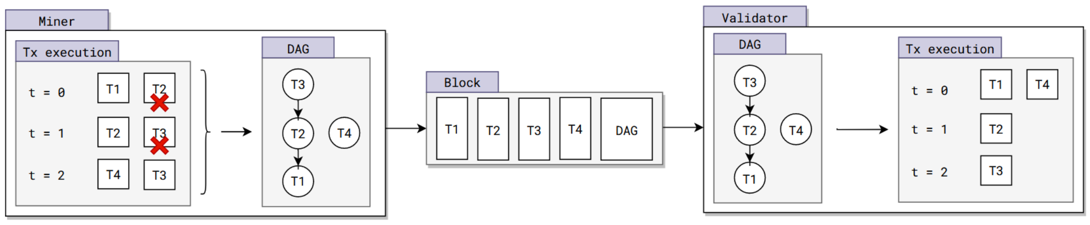


Figure 3.1: Dependency graph workflow.

the miner uses locks and tries to execute transactions concurrently, rolling back transactions that lead to conflicts. Conflicting transactions are then placed in a sequential bin, while non-conflicting transactions are placed in a parallel bin. After executing all non-conflicting transactions, the miner starts the second phase, executing all transactions from the sequential bin one by one. Although this approach avoids the security and space issues of appending the DAG capturing the dependencies in the block, it fails to harness the maximum concurrency in the block, as some transactions placed in the serial bin could still be parallelized.

OptSmart [15] provides a similar strategy, relying on Multi-Version Timestamp Ordering Software Transactional Memory (MVTO STM), which maintains multiple versions for each transactional object (t-object) reducing the number of aborts. The miner executes transactions optimistically and non-deterministically under the protection of STM, and when a conflict is detected, it is rolled-back and re-executed, and the conflict is captured on a Block Graph (BG) using an adjacency list structure. Transactions that do not conflict are stored in a concurrent bin. The BG and the concurrent bin are sent piggy-backed in the block, and the validator then runs the transactions in the concurrent bin concurrently, and only after it executes concurrently the transactions following the captured conflicts in the BG. This strategy takes the previous idea of separating conflicting from non-conflicting transactions [17], while improving performance and space. By using a BG to capture dependencies, validators can achieve better concurrency when executing the transactions in the block. On the other hand, by only storing conflicting transactions in the BG, the block uses less space as non-conflicting transactions are not represented, getting rid of unnecessary nodes in the graph. Still, this approach suffers from the same problems as the previous works. Although theoretically the block may not even include the BG (in the optimal case there are no conflicts), in practice, this is unlikely, as conflicts are predominant in the blockchain environment [1, 2, 17]. Thus, suffering from security issues (high probability of appending the BG in the block), as well as higher latency to send/receive the block (although with less impact than the previous works).

BLOCK-STM [14] solves these problems, avoiding the need of a dependency graph sent along the block, by ensuring deterministic execution at all nodes. Their approach relies on an Optimistic Concurrency Control (OCC) scheme along with STM execution with deterministic commit order according to the order transactions appear in the block. Miners execute transactions optimistically, recording the

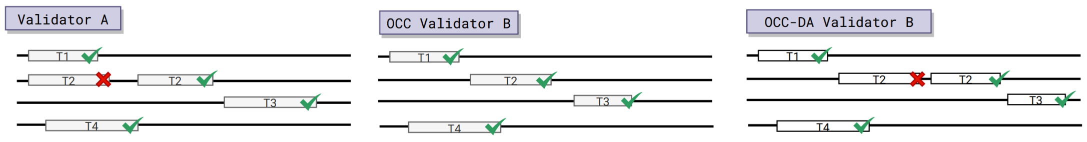


Figure 3.2: OCC with deterministic commit order vs. OCC-DA. **adapted from:** [1]

RWS at runtime. While a transaction is in progress, its reads can access the storage state corresponding to the highest transaction that occurred before the current one. After the transaction finishes execution, it undergoes a validation phase. During this phase, the transaction re-reads the previously recorded RWS to verify the validity of all the values involved. Successful validation alone does not guarantee commitment; therefore, a transaction can only be committed once all preceding transactions have undergone validation and have been committed.

The idea of executing transactions in a predetermined order was first explored in database systems like BOHM [27] and Calvin [28], where transactions execute deterministically using pre-known write-sets. However, this approach is impractical in blockchains due to challenges in predicting the complete RWS for smart contracts. BLOCK-STM captures the RWS at runtime, avoiding the need to know it beforehand. Although it may incur up to 30% overhead in sequential execution, their solution introduces two new improvements - deterministic execution and transparency. For the former, since they impose additional constraints on transaction commit order, they achieve determinism without relying on any additional structure to force the same equivalent serial order. As a result, their proposal is almost completely transparent to the current system, only requiring changes in the execution phase. The adoption of concurrent systems is facilitated by this approach, as it minimizes costs. There is no necessity to modify other components within the system, and there is no need to redesign existing components to address any security issues that may arise when employing a DAG.

Garamvölgyi et. al [1] raised a concern related to the usage of OCC with deterministic commit order as we have seen in BLOCK-STM. This strategy, although guaranteeing the same final state on all nodes, cannot guarantee the same execution process, i.e., cannot guarantee that if T1 aborts at node A, T1 also aborts at all other nodes. Optimistic Concurrency Control with Deterministic Aborts (OCC-DA) [1] aims to solve this issue. As an illustrative example, refer to Figure 3.2, where two distinct validators execute the same set of transactions with an equal number of threads. Validator A concurrently executes T2 with T1, leading to an abort of T2 due to a conflict with T1. T2 is subsequently re-executed. Now, consider OCC Validator B. In this scenario, T2 is only executed after the completion of T1, influenced by non-deterministic events on the thread handling T2. Consequently, T2 proceeds without any abort. This can become problematic in systems that rely on commit/abort decisions to penalize aborts to avoid DoS attacks [1]. As such, two different nodes may charge different amounts on the same user account depending on the number of aborts. The proposal aimed to solve this issue is a system based of OCC with **snapshot isolation** where transactions are committed according to their order in the block, but further refined to also produce deterministic aborts. This guarantees that if a transaction is aborted once on one node, it is aborted exactly once on all other nodes. To achieve this, each transaction is assigned a **storage version**, which specifies the transaction version up to which the storage writes can be accessed. Meaning a transaction can only see writes on storage up to the transaction specified by the storage version. This also means a transaction can only see these values after the corresponding storage version transaction has committed. If it is not committed yet, the transaction cannot start and must wait. Upon a conflict, the storage version is set to the highest transaction lower than the one executing. As an example, consider Figure 3.2. In the traditional OCC with deterministic commit order, Validator A and OCC Validator B both achieve the same final state, even though their execution differs. On the other hand, assuming a storage version of 0 was set at transaction 2, OCC-DA Validator B will abort T2 even if it is executed after T1, as it cannot read the values written by T1, and will detect the conflict when validating. It is then re-executed and sets the storage version to 1, committing this time. Since the value of the storage version impacts performance, two strategies are employed to predict the storage version statically and deterministically: **gas limit** - transactions with higher gas limit depend on transactions with lower gas limit in the assumption those with higher gas will finish executing after, and static analysis to get an approximate dependency graph. Which need not be perfect, but rather deterministic. These restrictions increase the execution overhead slightly, but solve the non-determinism of aborts, presenting a good trade-off between performance and security. Furthermore, it also limits wasted resources on re-execution. Besides OCC-DA, Garamvölgyi et. al [1] also proposed the use of **partitioned locks**. The idea comes from the study that shows most conflicts are usually bound to single atomic counters, used mostly under increment operations. To improve concurrency, the authors propose partitioning these counters into sub-counters, reducing conflicts on write-intensive workloads. The overhead appears only on read-intensive workloads, since each read will need to sum all sub-counters, conflicting with all current writers. Nevertheless, the study shows that the speedup almost doubled when using 2 sub-counters. The last strategy proposed by Garamvölgyi et. al [1] is devising special cumulative addition operations, tailored to deal with concurrent read-modify-write operations, and eliminate conflicts on transactions that issue only increments without using the read value.

So far, all the presented works rely on the same coarse grained concurrency granularity at the transaction level. Lin et. al. [2], as well as other empirical analysis [1, 17] on Ethereum blockchain have shown that neither this coarse granularity nor traditional concurrency control algorithms are suited for blockchain workloads. Thus new mechanisms must be developed with finer granularity. Lin et. al. [2] introduced a strategy that operates at operation level. Nodes execute transactions similarly to any other OCC scheme, but, at conflicts, only individual EVM operations that depend on conflicts are re-executed. Their proposal relies on capturing the dependencies of each operation dynamically at runtime, and record

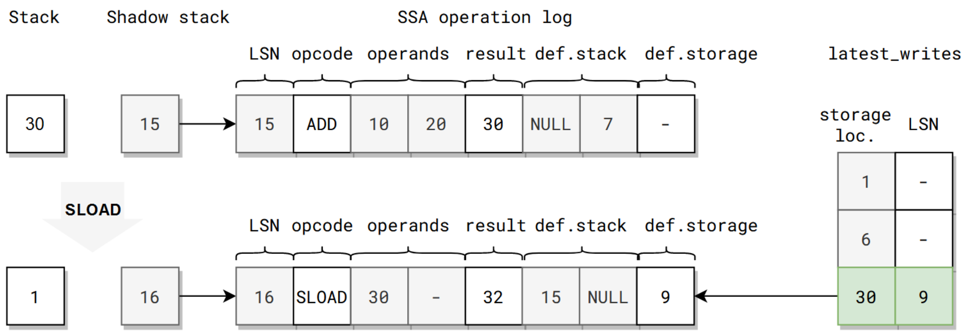


Figure 3.3: Olive [2] stack and storage Single Assignment Log (SSA) generation.

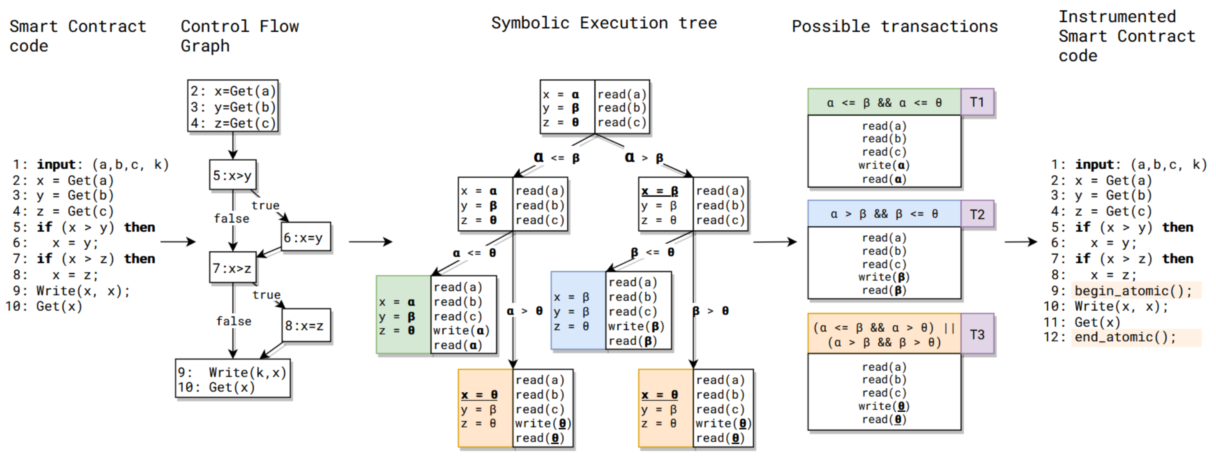
them on a Static Single Assignment (SSA) operation log. This log tracks dependencies of operations related to both stack, memory and storage (mainly related to `SSTORE` and `SLOAD` operations, described in Chapter 2). This is achieved by inserting a new entry for every EVM operation, recording the operation, the operands, the result, and the log entry from which the operands depend on.

Consider the scenario illustrated in Figure 3.3. Suppose a preceding stack operation, specifically `ADD`, had generated the value 30 within the stack (top stack entry). Upon executing a subsequent `SLOAD` operation (bottom stack entry), the associated storage location is scrutinized to ascertain any preceding writes conducted by the ongoing transaction. In this instance, a prior write, denoted by the SSA log sequence number 9 (highlighted in green), had written a value into the storage item that the current transaction is presently reading (marked under the first operand, with value 30). Consequently, this log entry (9) needs to be tracked by the `SLOAD` operation's entry - precisely, assigning sequence number 9 under `def.storage` within log entry 16. Besides this storage dependency, a stack dependency persists, as entry 16 references the value 30 derived from the previous stack operation with sequence number 15. Thus, the `SLOAD`'s log field `def.stack` is set to 15.

When re-executing, the system relies on special entries created at control-flow operations - **constraint guards** - that ensure the same path is taken when re-executing. This strategy is much more tailored to blockchain workloads, with a much finer granularity that results, on average, on only 0.3% of the total transaction EVM operations to get re-executed in the redo phase [2]. Just as BLOCK-STM, their approach is transparent, focusing only on changes related to the actual execution phase, minimizing any efforts needed to implement such strategies in real-world systems. However, implementing operation tracking logs at stack, memory and storage level is not trivial in practice.

3.2 Static Analysis

Recent works seem to have an increasing interest on analysing smart contract code statically, to either instrument it, or to infer the RWS. Works following this strategy have the advantage of building information



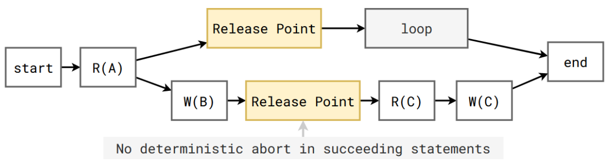
```
1: input: (a,b,c, k)
2: x = Get(a)
3: y = Get(b)
4: z = Get(c)
5: if (x > y) then
6:   x = y;
7: if (x > z) then
8:   x = z;
9: Write(x, x);
10: Get(x)
```

```
1: input: (a,b,c, k)
2: x = Get(a)
3: y = Get(b)
4: z = Get(c)
5: if (x > y) then
6:   x = y;
7: if (x > z) then
8:   x = z;
9: begin_atomic();
10: Write(x, x);
11: Get(x)
12: end_atomic();
```

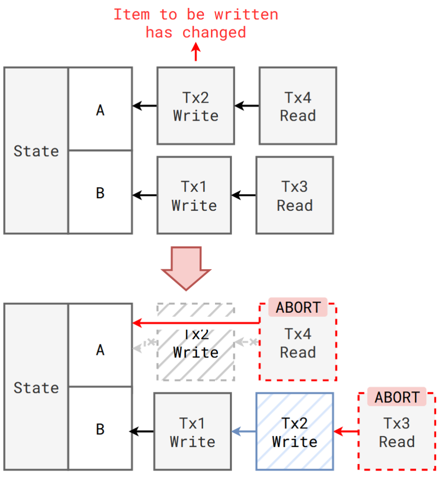
Figure 3.4: FastBlock Atomic Section identification.

that help scheduling transaction execution based on the contract code. However static analysis can still be inaccurate. Li et. al [29] proposed a strategy that combines static analysis with concurrent execution under HTM support. Miners identify Atomic Sections (AS) and their conflicting conditions - statements specifying, for each method and each input parameter, which ASs conflict - using symbolic execution. The workflow is shown in Figure 3.4. A smart contract is first decomposed into a Control Flow Graph (CFG) representing the possible paths the program can take. Then a symbolic execution tree is built following the CFG, and the RWS is captured while the paths are being parsed. Paths that end up with the same RWS are considered to be equivalent, and their path constraints are joined as explain in Section 2.3. Then, the RWS of each of these transactions is compared against each other to identify all possible conflicts assuming the worst case. Finally, instrumentation primitives are inserted in the contract code.

After the instrumentation, miners execute the transactions and code within an AS is executed under HTM protection. When conflicts are detected, the corresponding AS is aborted and re-executed. Miners then produce a commit order based on the CPU recorded timestamps, and broadcast the block. Validators then produce a graph at AS level from the serial block order, and repeat the same process as the miner. Note that the instrumentation only needs to happen once, when the contract is first received. This approach takes advantage from static analysis to produce finer granularity - at the level of ASs, instead of transaction level. However, relying on HTM comes with some drawbacks. Namely limited support for diverse architectures, and also no guarantees the AS will ever commit. This requires devising a fair execution cost charging mechanism, such as considering only the cost of the last execution. However, doing so, we allow malicious nodes not to be penalized for proposing highly conflicting transactions - a problem raised by Garamvölgyi et. al [1], as discussed earlier in Section 3.1. Also, the insertion of additional instrumentation primitives in the contract code raises the total transaction execution cost, as these have to be also executed under the EVM. Thus, the approach, although performant, lacks portability and transparency, as it requires specific hardware, and incurs additional costs for the users.



(a) Partial State Access Graph (SAG).



(b) Dependency tracking with dependent reads.

Figure 3.5: Qi et. al. proposal. **adapted from:** [3]

Qi et. al. [3] propose using static analysis via symbolic execution to capture all possible RWS for every input by analyzing each smart contract's code when it is first received by a node. Upon receiving a new transaction, nodes use the pre-constructed State Access Graph (SAG) (see Figure 3.5(a)), or compute it on the fly, to predict the data store items that will be accessed by the transaction, and place the transaction along with its graph in the memory pool. As some transactions may depend on values from state, some nodes in the SAG cannot detect the state item to be accessed, and a read to the current blockchain snapshot may be needed to complete the SAG. Miners then track the access operations (reads and writes) of each transaction per storage item, keeping the order of transactions that appears in the block, as depicted in the top side of Figure 3.5(b). This list-like data structure implements multi-versioning, ensuring no write-write conflicts are possible, thus minimizing conflicts. Transactions with no dependencies are first picked to execute, and those that need to read from a storage operation from a previous transaction must wait until all dependent storage items are written. To reduce coarse granularity, an early-write visibility strategy introduces Release Points in the SAG, allowing transactions to make their written values visible to others while ensuring correctness. These Release Points ensure that (1) no other write occurs afterward, and (2) the transaction will not abort by any deterministic abortable statement, achieved by predicting an upper bound on gas costs of the remaining execution. This strategy increases performance by refining granularity.

The main drawback of this strategy is notable when the prediction of the RWS is not accurate. Following the example in Figure 3.5(b), when a transaction is executing, and a RWS that was not captured by the SAG is detected, the access sequence must be updated, and transactions depending on either the wrongly predicted state item (`Tx4`), or on the highest transaction that is lower than the one executing which wrote to the new state item (`Tx3`) must be aborted. This leads to **cascade aborts**, recursively aborting transactions and hurting performance. Besides cascading aborts, the strategy also presents a less-optimal decision, by computing the SAG upon a transaction is received, and storing it in the memory pool. Due to the workflow of blockchains, this may lead to long waits in the pool, making the predicted RWS outdated, as it was computed from an increasingly older state, ultimately leading to an increase in the number of cascaded aborts.

Both works present improvements in terms of transaction granularity when compared to the majority of speculative execution's works.

3.3 Smart Contract Isolation: Intra- vs. inter-SC conflicts

As discussed in Section 2.1.1, smart contract's storage is isolated from one another. The works on this section schedule their execution based on this idea of isolation, and assign contracts to threads/nodes based on the contract's address, with the goal of minimizing synchronization costs across threads/nodes.

ParBlockchain [30] proposes OXII, an **order-execute** paradigm for deterministic parallelism. It consists of **orderers** to establish transaction order and generate a dependency graph without executing the transactions, based on prior knowledge of transaction's RWS, either predeclared, obtained via static analysis, or speculative execution. **Agents** then install the corresponding smart contracts based on the contract's address, and when executing, only the contracts for which the node is an agent of are executed by the node. This allows for inter-node concurrency, as different agents can execute different contracts concurrently. However, when cross-contract dependencies arise, nodes must wait on the result from the corresponding agent, thus hurting performance, as the inter-node communication is expensive. This assignment of contracts to specific nodes is possible only in permissioned environments, where the identity of the participating nodes is known.

FISCO-BCOS [31] tackles performance optimization at two different levels: inter-block and intra-block. To improve inter-block concurrency, they propose a Block Level Pipelining (BLP) strategy composed of four stages - Ordering, Execution, Checkpoint, Commit - where each stage may be parsing a different block concurrently. The proposed BLP dynamically adjusts the number of blocks being proposed depending on execution speed, and utilizes in-memory states that record the execution of a previous executed but uncommitted block, such that new executions may start, and rely on previous state. For intra-block concurrency, FISCO-BCOS devises a similar strategy as ParBlockchain [30], creating a scheduling based on smart contract addresses, but differently from [30], this is done at the thread level. Smart contracts are assigned to different thread pools, based on the idea that conflicts are reduced across contracts within the same pool. Still, cross-contract calls across different pools require synchronization barriers which incur overhead as it must wait on all other executors to finish processing transactions. Both FISCO-BCOS and ParBlockchain are specific to permissioned environments, limiting their applicability. Furthermore, although the idea of isolating smart contract execution is intuitive, there may still be conflicts. Even if no cross-contract dependencies exist, when two contracts change global state such as the balance of the same user account, they conflict. Such cases have not been explicitly addressed on these works.

SC-CHEF [32] proposes decomposing transactions into multiple sub-transactions that can be executed independently while respecting their logic dependency. Each sub-transaction corresponds to a call to a smart contract, and is mapped to a corresponding thread based on the data it accesses. Upon receiving a new sub-transaction, a new node is created in the local dependency graph captured by the corresponding thread. After the graphs have been built on all threads, each thread executes the sub-transactions following the graph's topological order, and respecting cross-contract call dependencies. Differently from the previous two works, SC-CHEF can be employed in a permissionless environment, but lacks a devised strategy on how to construct the RWS. They assume the RWS may be given by the clients, which is not a practical solution. Also, as in the previous two works, SC-CHEF suffers from slightly increased cross-thread concurrency due to conflicts on global state, not solved by contract isolation.

All these works provide finer granularity when compared to most of the previous works discussed. Still, although exploiting the idea of contract isolation to minimize thread/node synchronization, they suffer from the hot-spot problem. Since a large amount of calls is made to a small number of threads/nodes, this ultimately leads to specific threads/nodes being overloaded while the others are underloaded, resulting in an unbalanced utilization of resources.

3.4 Discussion

Several proposed solutions address the problem at hand but often lack optimal design for hot-spot workloads. Approaches requiring a graph to be sent with the block [15, 16] suffer from transparency and security issues, while being unsuitable for high contention due to coarse transaction granularity, resulting in significant resource wastage due to their optimistic nature. Solutions using OCC [14] also incur high costs during transaction aborts. Saraph and Herlihy [17] limit aborts per transaction but do not parallelize conflicting transactions. Conversely, OCC-DA [1] reduces aborts and allows parallel execution of conflicts, enhancing security and resource efficiency at low overhead. Olive [2] improves upon this by applying operation granularity, only re-executing specific operations, which lowers abort overhead but still incurs costs from constructing the Static Single Assignment (SSA) log even in the absence of conflicts.

We have also seen solutions specific to permissioned environments [30, 31] that introduce finer granularity at contract call level, but suffer from cross-contract calls across different nodes [30], and across different thread pools [31]. Due to the limitation of being specific to permissioned, we will not focus on the ideas of installing smart contracts in specific nodes, nor in devising an execution pipeline

Work	Abort-Induced Resource Waste	Roll-back overhead	Scheduling wait time overhead	Require Block Graph	Concurrency granularity	Requires RWS known a priori	Op Comm Support
Fine-grained State Access [3]	Dependent on number of DTs*	Very High	Medium-High (if last write is at end of tx)	Yes	Last Write / Transaction in worst case	No	No
FastBlock [29]	Potentially High	Low/Medium (depending on size of atomic section	None	No	Atomic Section	No	No
Dickerson et. al. [16]	High (bounded to number of txs	Miner-High Validator-None	Miner - None Validator - High	Yes	Transaction	No	No
BLOCK-STM [14]	Potentially high (unbounded)	Medium (reading ESTIMATE) / High (validation)	None	No	Transaction	No	No
OptSmart [15]	Potentially high (unbounded)	High	None	Yes/No (if there is no conflict)	Transaction	No	No
OCC-DA [1]	Low (at most 1 abort per tx)	High	None	No	Transaction	No	No
Olive [2]	Potentially low (unbounded)	Very Low	None	No	Operation	No	No
Speculative bin [17]	Low (at most 1 abort per tx)	High	None	No	Transaction	No	No
ParBlockchain [30]	None	None	Medium / Very high (cross-contract calls)	Yes	Contract call	Yes	No
FISCO-BCOS [31]	None	None	Medium / High (cross-contract calls)	No	Contract call	No	No
SC-CHEF [32]	None	None	Medium	No	Contract call	Yes	No
COEX-P	None	None	Low-Medium	No	Operation	Yes / No	Yes

Table 3.1: Comparison table of discussed works.
*DTs (Dependent Transactions) refers to transactions whose RWS depends on some state value.

as proposed by these works. SC-CHEF [32] takes a similar approach as the previous two works, but is not limited to permissioned systems. Still, as discussed in Section 3.3 it still suffers from the hot-spot problem, causing threads responsible for popular contracts to have most of the work assigned to them, causing an unbalanced resource utilization.
Fastblock [29] uses SE to detect ASs, but causes the cost of transaction execution to increase, while still suffering from potentially high resource waste due to AS re-execution, as HTM cannot guarantee it will commit. Qi et. al. [3] proposes a less-constrained and more transparent solution, with low resource waste, but still suffers from cascading aborts upon RWS mispredictions. It also suffers from path explosion, which need to be solved. One solution is to read the part of the transaction that could not be fully predicted, but this leads to overhead as reading state is expensive. Still, the solution improves concurrency granularity.

Many of the suggested strategies operate at a transaction granularity, which may not be ideal. In scenarios with substantial contention, common in typical workloads, this approach leads to costly aborts or prolonged waiting times, depending on the chosen strategy. Works that provide contract call granularity also suffer from the hot-spot problem as discussed above.

Table 3.1 provides a comprehensive comparison of the discussed solutions alongside our proposed approach. The assessment involves a comparison of abort-induced resource waste, indicating the extent of resource waste attributed to aborts throughout the entire execution. It's important to note that this metric depends not only on the overhead of a single roll-back but also on the total number of roll-backs. Roll-back overhead is the overhead associated with individual roll-backs, where coarser granularity results in higher roll-back costs. Scheduling wait time overhead quantifies the time wasted while waiting for another transaction to complete without contributing useful work. This is particularly relevant in scheduled executions. Solutions are further classified based on their dependency on a block graph for tracking dependencies and their assumption about the knowledge of abort-induced resource waste in advance. Finally, Operation Comm. Support refers to the ability to support and benefit from commutative operations in the workload.

Additionally, we categorize the granularity of transactions in the system into five levels: **Transaction**, **Atomic Section**, **Last Write**, **Operation**, and **Contract Call**. Our approach aims low to medium scheduling wait time overhead during execution, ensuring efficient handling of high-contention scenarios. By addressing the hot-spot directly, integrating commutative operation support, we propose a scalable strategy that optimizes performance for workloads involving frequent conflicts, particularly those with numerous commutative operations across a small number of smart contracts.

The solution leverages granular concurrency control at the operation level to further enhance parallelism. Notably, for the **RWS known a priori** entry, we have designated the value as `Yes / No`. This reflects that, although our system supports the prediction of RWS based on a symbolic execution tree for a given smart contract, we do not currently support dynamic changes to the RWS during runtime. In essence, we either assume that the RWS of a smart contract is independent of its state—allowing us to parse the symbolic execution tree to predict the RWS—or we receive the RWS information beforehand.

4 COEX-P

This Chapter presents **COEX-P** (Commutative Operation EXecution in Parallel), a framework for parallelized smart contract execution implemented in CosmWasm execution environment for the Cosmos ecosystem. The solution is designed to address the inherent performance limitation of current serial execution environments under hot-spot workloads by leveraging symbolic execution for conflict detection and operation commutativity identification, and an optimized scheduling approach for transaction parallelization.

Although developed for the Cosmos ecosystem, using CosmWasm as the default VM, the solution is not environment-specific and can be adapted to other architectures, such as Ethereum.

The rest of this Section is structured as following. Section 4.1 details the execution model and some of its limitations. Section 4.2 provides an overview of the overall architecture, and its workflow. Section 4.3 describes commutative operations and how they are supported by the system. Section 4.4 details the inputs for the system provided by symbolic execution, and how they are parsed and utilized by the system. Section 4.5 describes thoroughly the process for creating the schedule, and how it addresses the optimizations described in the preceding section. In Section 4.6 we detail the runtime updates made to the schedule during execution, and how synchronization is achieved. Finally, Section 4.7 describes how the final schedule is persisted into the storage, and how it ensures this persistence respects the block's transaction order.

4.1 Cosmwasm Execution Workflow

As previously described in Section 2.1.2, the CosmWasm execution model delegates the execution of smart contract logic to the WVM while handling critical external operations such as storage access and cryptographic functions within the CVM. This separation allows WVM to focus solely on business logic, whereas CVM manages the contract's interaction with the blockchain state through external calls.

This limits the control over the execution, as strategies such as the ones proposed by Lin et. al. [2], taking advantage of stack-level operation control, become unfeasible to implement as this granularity is abstracted from the CVM, and only available within the WVM. As such, the proposed solution works with these read/write delegations to the CVM, using the information passed within those methods to synchronize the actual execution with the predicted schedule as we describe in the following sections.

4.2 System Architecture Overview

We first briefly describe the key system requirements, and then present a top-level overview of the system.

The proposed solution presents a novel approach to parallelizing smart contract execution based off of three base requirements:

- **Transparency:** Our approach preserves the existing blockchain security model by avoiding modifications to its core infrastructure. This minimizes the risk of new vulnerabilities and ensures that the original security guarantees remain intact, maintaining the system's integrity and trustworthiness while optimizing smart contract execution.
- **Performance:** By leveraging conflict-aware parallel execution of smart contract transactions, the system can achieve higher throughput.
- **Hot-spot awareness:** By relying on commutative operation identification via symbolic execution, the execution can further parallelise commutative operations aimed for the same smart contract storage item.

COEX-P mainly operates within the smart contract execution phase of the blockchain process, right after a block has been constructed from a set of transactions waiting in the memory pool. It does not introduce changes into the consensus, nor the block itself as some of the previously discussed works did.

Figure 4.1 describes the high-level overview of the system - The system works with two distinct entry points at two slightly different levels - smart contract **deployment**, and smart contract **invocation**. Smart contract deployment involves the deployment of WASM bytecode which is to be saved in the node. At this point, COEX-P relies on a symbolic execution engine to parse the received code to produce a **profile** for that contract. This profile is an in-memory structure that stores all possible execution paths of the contract, enabling the extraction of a concrete execution path based on specific inputs. It evaluates the relevant conditions at runtime, returning the RWS for the parsed path.

The specific implementation of this symbolic execution engine is beyond the scope of this Thesis; instead, we focus on defining an interface that captures the expected output from such an engine.

The other entry point is related to the block execution. When the node receives enough transactions either invoking a method from an already-deployed contract, or instantiating a brand new smart contract from a previously deployed WASM bytecode, a block is built, and sent to the **VMManager**, which controls the overall execution phase, as shown in more detail in Figure 4.2. For each transaction in that block, we

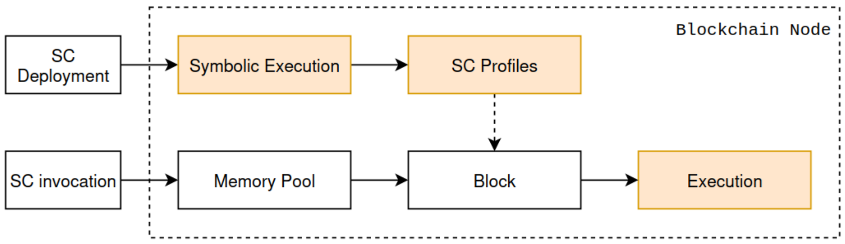


Figure 4.1: COEX-P workflow

compute its RWS from the smart contract's profile, and use this information to schedule the transactions, and execute them in parallel.

The execution phase itself can be further divided in three sub-phases, as we describe in more detail in the following sections:

- **Schedule Creation:** This phase is concerned with gathering all RWS information, and using it to detect the conflicts between transactions, and schedule them accordingly in a way that ensures deterministic execution.
- **Schedule Execution:** Given a schedule, this phase is concerned with executing the transactions following the input schedule. It is at this phase that thread management and runtime concurrency control are ensured, making sure threads do not execute conflicting operations, execute balanced workloads, and that they update the schedule accordingly.
- **Schedule Persistence:** After the execution phase has taken place, this phase runs over the final schedule and uses its information to persist the results of execution in the smart contract's storage.

Figure 4.2 illustrates all three execution phases in more detail. As explained earlier, the CVM works as a wrapper around the WVM, and stores the execution context from the blockchain application perspective. This means in practice that it is the CVM that keeps the **schedule** data structure, the **RWS** information, as well as the smart contract **storage**. Once a thread starts executing, it delegates the execution to the WVM, which in turn may invoke CVM functionality. The two most relevant imported calls in the WASM bytecode are `db_read`, which reads from storage, and `db_write`, which writes to storage. In practice, each of these calls needs access to the three data structures held by the CVM, as we will describe later on. Once the execution is finished, the WVM is discarded, and the thread will be assigned another transaction to execute, with the corresponding context for the associated smart contract at hand. It is important to also refer that each CVM is responsible for a single WVM, and ech WVM needs a bytecode as input. This means there will always need to be at least one CVM for each smart contract, initialized using that contract's bytecode.

When all transactions have been executed, the final schedule is iterated over, and the last written

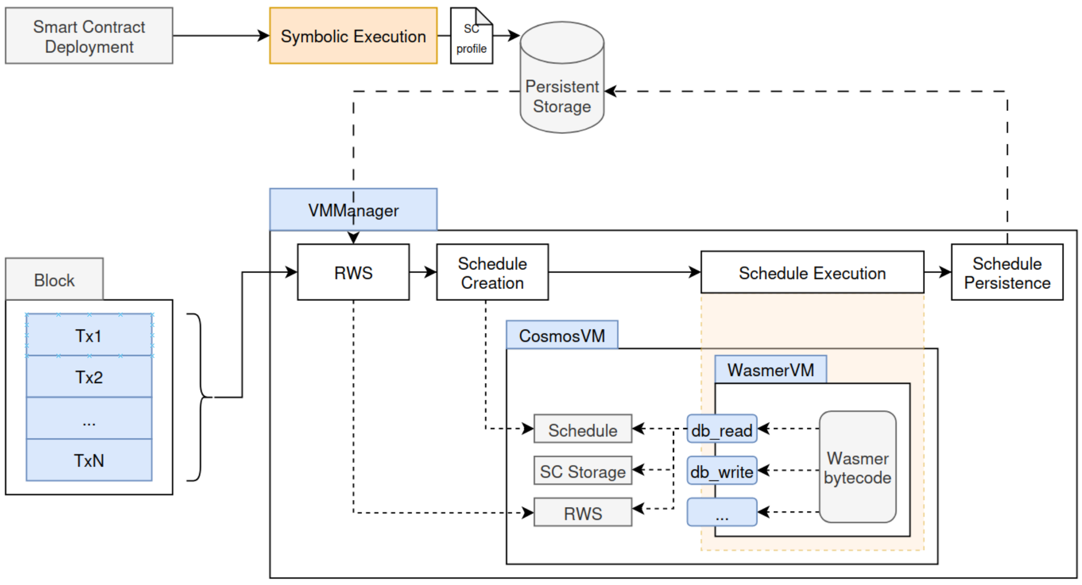


Figure 4.2: VM execution workflow.

values for each storage item are persisted into the smart contract's storage, which is then persisted into the node's persistent storage.

In our proposed solution, we make a key **assumption** to simplify implementation and focus on showcasing the benefits of commutative operations: we assume that the RWS of each smart contract is determined once during the scheduling phase and remains unchanged throughout execution. This means that **once the RWS is predicted, the smart contract cannot issue any new read or write operations that were not anticipated**. In essence, the system operates under the assumption that there are no dynamic updates to the RWS, even if the actual workload could vary based on the current state of the contract's storage. We will discuss the consequences of this limitation later on Section 4.4.

4.3 Commutative Operation Support

Before delving into the solution's components in more detail, we first introduce the concept of operation commutativity and its impact on the system's design. Later, in Section 4.6, we explain in more detail how exactly the system parallelizes commutative operations.

Operation commutativity refers to the property of certain operations where the order of their execution does not alter the final state. Formally, two operations O_1 and O_2 are said to commute if applying them in different orders yields the same result, i.e., $(O_1 \circ O_2)(x) = (O_2 \circ O_1)(x)$. This concept plays a crucial role in concurrent systems by allowing operations to be executed in parallel without violating correctness, thereby improving performance by reducing the need for strict serialization.

Commutative operations can practically be decomposed into basic read and write operations. For

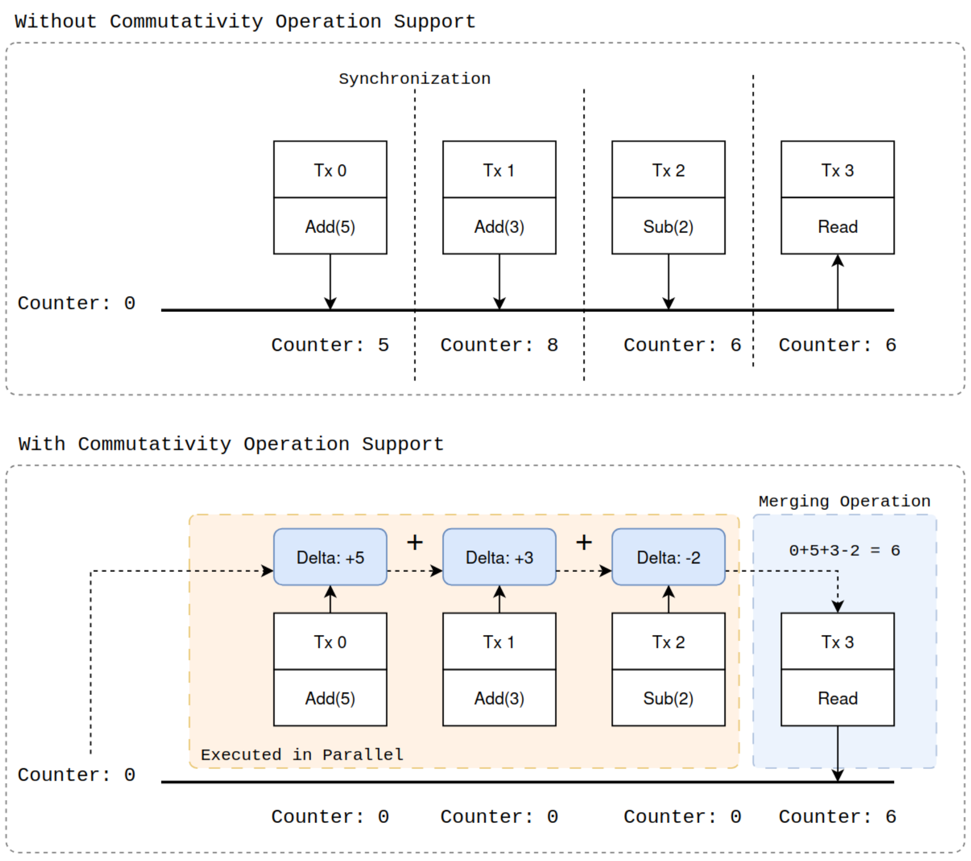


Figure 4.3: Operation Commutativity Overview.

instance, consider an incremental commutative operation such as $A = A + 1$. This operation can be expressed in terms of reads and writes as $W(A) : R(A) + 1$, where a read is performed on item A, followed by a write of the incremented value. **We rely on the symbolic execution engine to identify the read and write operations associated with any abstract commutative operation** (e.g., increment) allowing the system to perform those operations in an order-independent manner.

This commutative property is especially useful when performing **parallel execution**, as illustrated in Figure 4.3. Operations that commute can be applied concurrently in any order, and their effects captured as **deltas** — incremental changes that each operation makes to the shared state. For instance, consider a counter A: multiple increments on A can be treated as deltas (e.g., $+1$) rather than requiring immediate updates to the shared variable. Similarly, in data structures like sets, additions or removals can be handled as deltas without needing to directly modify the structure until the final merging phase.

As previously mentioned in Sections 2.1.2 and 4.1, the system synchronizes execution based on read and write operations. As such, we must define commutativity in terms of these operations. **Commutative read** refers to a read operation associated with an incremental write (such as adding a *delta* to a value). Unlike typical reads, a commutative read does not require the most up-to-date value of the storage item. Instead, it captures an initial reference value, which will later be used to calculate the effect of the incremental write. This process will be explained in more detail in Section 4.6, when we explain how, during execution, we support parallel execution for commutative operations. Take as an example the following incremental write $W(A) : R(A) + R(B)$ that needs to perform a read on A, and a read on B. We define $R(A)$ as commutative since it is directly associated with the incremental write in the sense that it reads the exact same key being incremented. As such, this read does not require to read most recent state, but can read any previous value for the purpose of computing the increment. $R(B)$, on the other hand, is a **non-commutative read**, as it is part of the increment on item A, and as such, requires reading the most recent value from B. Writes are also characterised in a similar manner. In the previous example, $W(A)$ is said to be a **commutative write** since the written value is an increment over A's previous value. If, on the other hand we had the operation $W(A) : R(B)$, the write would be classified as a **non-commutative write**.

The performance benefit emerges when **merging** these commutative operations. After running in parallel, the system efficiently combines the deltas into a final result, merging all the deltas to the initial value. The only synchronization point is now at the final state, after having all deltas.

This approach avoids the complexity of handling conflicts in commutative operations, which would otherwise require more rigorous synchronization and strict ordering.

4.4 Profile Generation

As outlined in Section 4.2, the first phase of the proposed solution is initiated upon the deployment of a smart contract. During this phase, the SE engine performs an in-depth analysis of the contract's WASM bytecode, producing a structured output that encapsulates all potential execution paths, conditional branches, and the corresponding RWS operations for each path. This output is critical as it provides a detailed representation of the contract's possible behaviors and associated state modifications.

Listing 4.1 shows an example of the `Execute` entry point of a simplified CosmWasm smart contract. The entry point is declared in lines 1-10, and the inputs of the entry point, which will be used later for parsing the contract's profile are declared in lines 2-5. CosmWasm contracts follow a convention of matching the received message (`_msg`) against all possible `enum` variants. The example also follows this practice, and we can see the matching in lines 7-9, where we extract a field `user` within the message, and use it to call the `add()` function. This function is also simple. It fetches a user given its identifier, and increments its balance by one.

```
pub fn execute(
    _deps: DepsMut,
    _env: Env,
    _info: MessageInfo,
    _msg: ExecuteMsg) -> StdResult<Response> {
    use ExecuteMsg::*;
    match _msg {
        AddOne { user } => execute::add(_deps, user),
    }
}
...
pub fn add(deps: DepsMut, user: String) -> StdResult<Response> {
    COINS.update(deps.storage, user, |bank: Option<i64>| {
        match bank {
            Some(value) => value + 1,
            None => Err(StdError::generic_err("Value doesnt exist")),
        }
    })?;
    Ok(Response::new())
}
...
```

Listing 4.1: Smart Contract function example.

Upon receiving the compiled contract, the SE engine processes the bytecode and generates a serialized output, which follows a structured format. Given that CosmWasm contracts are organized into distinct entry points (e.g., `Instantiate`, `Execute`), the output produced by the SE engine will be similarly segmented. This ensures that the behavior of each entry point is captured independently.

The format generated by the SE engine follows a predefined structure, as we describe next. This format is segmented by contract entry points (e.g., 'I' for `Instantiate`, 'E' for `Execute`), and within each section, variables relevant to the transaction are explicitly listed. Additionally, path conditions (e.g., `PC_1`, `PC_2`) identify execution branches, accompanied by storage operations such as `GET` and `SET`, which may be dependent on transaction inputs (e.g., `GET(=AARiYW5r @ _msg.user)`, where `AARiYW5r` stands for the base key bytes in base64, `_msg.user` is the value from the transaction input, and `@` appends the right side to the left). We rely on the SE engine capabilities to annotate each read and each write with a flag indicating its commutativity status (`Inc` for commutative, `Non-Inc` for non-commutative). This differentiation is essential for enabling parallelization during transaction execution.

```
...
E ----------------------------
_deps: DepsMut
_env: Env
_info: MessageInfo
_msg: ExecuteMsg
> AddOne:
    - user: string

[PC_1] Type(_msg) == AddOne
=> GET(=AARiYW5r @ _msg.user): Inc
=> SET(=AARiYW5r @ _msg.user): Inc
<- None
...
```

Listing 4.2: Symbolic Execution output example.

Continuing our example, Listing 4.2 demonstrates the corresponding output from the SE engine for the previously discussed smart contract. The output begins by specifying the `Execute` entry point (line 2), followed by the expected inputs `_deps`, `_env`, `_info`, `_msg` (lines 3-6), and the message types supported (lines 7–8). Each different message type appears after the > sign. As we can see, only one message type is supported, with an argument `user` of type `string`.

Line 10 in the output identifies the path condition `PC_1`, which asserts that the input variable `_msg` must be of type `AddOne`. Below this condition, the `GET` (representing a read) and `SET` (representing a write) operations are listed, along with the storage keys they access. The commutativity identifier is set to `Inc`, indicating that the operations are commutative. Following the profile, right arrows (=>) indicate the positive branch, when the condition returns `true`, and left arrows (<-) indicate the negative branch, when the condition returns `false`.

This textual output is then converted into an in-memory tree structure, referred to as the contract's **profile**, representing the SE-generated symbolic output. This profile is stored persistently within the node's storage and linked to the smart contract's bytecode. Notably, the profile is static throughout the blockchain's lifetime; it remains immutable and is used to predict the RWS of transactions based on their inputs, and possibly on storage. This static profiling occurs, as explained, at contract deployment, and the profile remains unchanged, allowing it to be parsed when necessary to retrieve execution path predictions without requiring modification during the contract's execution.

Figure 4.4 illustrates the resulting in-memory profile structure resulting from parsing the SE output from Listing 4.2. The parent node in the symbolic tree is a `ConditionNode`, representing a conditional expression that is composed of three elements: a left-hand side (`lhs`) expression, a relational operator (`rel_op`), and a right-hand side (`rhs`) expression. Expressions can be further decomposed into sub-expressions, which may involve arithmetic operations, identifiers, transaction inputs, and more. For example, the condition shown in the figure (top-left purple node) checks if the input message type is `AddOne`. In this case:

- The `lhs` expression refers to the message type.
- The relational operator is `Equal`.
- The `rhs` expression is the custom type `AddOne`.

When parsing this profile, the condition is evaluated based on the transaction input captured in msg. If the condition evaluates to true, the system follows the `pos_branch`; otherwise, it follows the `neg_branch`.

Each node type in the symbolic execution tree illustration is represented by a distinct color, with blue nodes indicating one of two possible types:

1. **ConditionNode:** Represents a conditional check (as explained earlier), which evaluates expressions involving transaction inputs, storage values, or other dynamic factors.

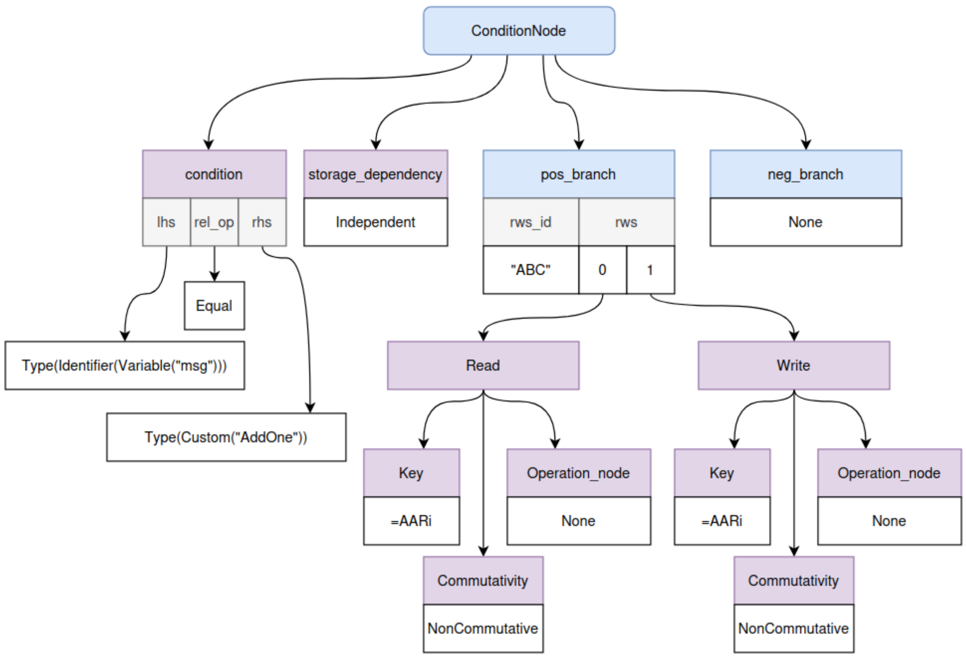


Figure 4.4: Smart contract profile tree structure.

2. **RWSNode:** Represents a leaf node containing the ordered sequence of read/write operations for a specific execution path.

In the example, once the condition is evaluated as `true`, the system follows the `pos_branch` and retrieves the corresponding list of RWS operations (illustrated by the indices 0 and 1 under `rws`). The unique identifier for this sequence is stored in the `rws_id` field under the `pos_branch`.

The `rws_id` is a unique ID generated based on the sequence of reads and writes, as well as the keys that these reads/writes operate on, and also on the commutativity of that operation. Two execution paths with the same ID will always have the same operation type and commutativity type in case both operations are aimed at the same item. This is used by the system to possibly reorder transactions to improve performance of the system when commutative operation support is enabled, as we discuss in Section 4.6.

Once the profile tree structure has been created and stored, the resulting output is a sequence of structures known as `TxRWS`. Each `TxRWS` contains the following fields:

- **profile_status:** Specifies if the parsed profile is complete or incomplete. Profile completeness is defined as the total coverage of the possible execution paths and its respective RWS. If for some reason, as explained in Section 2.3, the engine is not able to cover all possible execution paths, then this field is set to `Incomplete`. Else, it is set to `Complete`. Although this field, since we assume we always know the exact RWS we do not use this field, since it provides context on the limitations of the SE engine, it can be useful for future improvements, as discussed in Chapter 6.

- **storage_dependency:** Specifies whether any conditions in the predicted execution path depend on storage values, meaning that the actual RWS may differ during execution based on the state of storage at that time.
- **rws_id:** The unique identifier for the predicted execution path.
- **rws:** The ordered sequence of read/write operations along the execution path.

This sequence of `TxRWS` structures is then encapsulated within another structure that records the transaction position within the block. This composite structure serves as the input for the subsequent execution step: Schedule Creation.

4.5 Schedule Creation

After having parsed the contract profiles for every transaction in the block, we enter the schedule creation phase. We start by explaining the single-threaded process, and then we introduce the parallelized version of it.

Given the inputs from the previous phase, the goal of schedule creation is to efficiently produce a schedule capable of supporting concurrent execution of transactions, providing an execution framework capable of both concurrency control support and performant execution.

The idea is similar to what Qi et. al [3] proposed, where we have a set of storage items per smart contract, and each captures the sequence of operations, each represented by an **operation node**, aimed at that item, following the order transactions appear in the block.

Running Example Let's start by showing an example of the high-level logic for the schedule creation, and then we delve into the details of the algorithm. Figure 4.5 shows an example of schedule creation. At the top half we have the block outputted from the previous phase as explained earlier. At the bottom half we have the final schedule form. At the left of the schedule stands the keys (`Key 1`, `Key 2`) of some smart contracts (`SC 1`, `SC 2`). The colored boxes at the bottom half illustrate the nodes representing the respective operations. Colors were used to differentiate different transactions. Dashed lines (with no arrow) represent the order of operations following block order, and filled arrows represent dependencies. We differentiate between two types of dependencies:

- **Explicit Dependency:** Represents a dependency between a non-commutative read and a non-commutative write. This type of dependency is **captured by the operation node**. Each node has, as will be explained in a moment, a `dependency` field that points to an operation the current node depends on. If this field is set, then we call that an explicit dependency, as it is explicit represented in the dependent node. Explicit dependencies are more easily thought as a barrier marking the

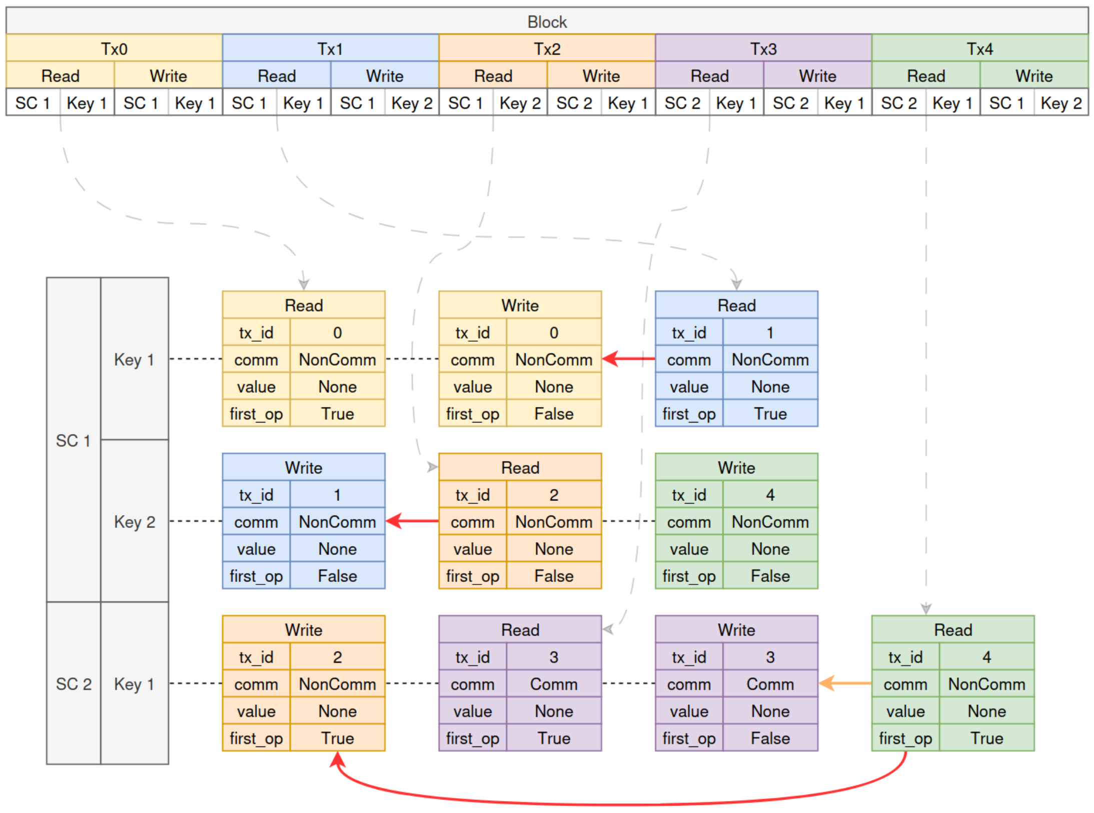


Figure 4.5: Schedule Creation Example. Red arrows represent explicit dependencies and orange arrows represent implicit dependencies. Black dashed lines represent the order of operations following block order. Light grey dashed arrows identify where in the schedule is the first operation for each transaction in the initial block.

base value from which all following commutative operations will be merged into. In Figure 4.5, an explicit dependency is marked between the green read at `SC2, Key1` to the orange write at that same key. This dependency marks that, as we will explain in Section 4.6, the commutative merging will use the orange node's value as the starting point, and will merge the purple commutative write at the right, returning the merged value to the green non-commutative read.

- **Implicit Dependency:** Represents a dependency between a non-commutative read and one or more commutative writes. This type of dependency is **not captured by the operation node** in the schedule, but by an in-memory structure. As such it is not visible from the schedule point of view, but it is still marked as a dependency, and the dependent operation will only be able to execute after this dependency has been met. This represents a dependency between a non-commutative read and any commutative write.

Explicit dependencies are represented by red arrows, and implicit dependencies are represented by orange arrows. Dashed arrows identify the first operation and the corresponding schedule node for each transaction.

The algorithm starts with the block of ordered transactions and their respective RWS, as represented in the top half of the illustration. We run over each operation of each transaction sequentially, following the transaction order in the block, and if it is the first operation for the specified smart contract, and the specified key, then we create an entry on the schedule, and create a brand new linked-list with only one node representing the current operation. For each operation there will be an **operation node** which will be created with the following information:

- **dependency:** Pointer to the explicit dependency of the node, if any.
- **next:** The next operation, either from the same transaction, or from another, following the transaction order of the input `BLOCK`.
- **prev:** The previous operation, either from the same transaction, or from another, following the transaction order of the input `BLOCK`.
- **tx_id:** The transaction index in the input `BLOCK`.
- **value:** The value read or written. This field is only set during execution.
- **first_op:** Boolean specifying if this is the first operation of transaction `tx_id`.
- **comm:** Describes the operation's commutativity. It is set to `Comm` if the operation is commutative, and `NonComm` otherwise.

After having created a brand new entry for the first operation of `Tx0`, we proceed iterating over all operations, and insert them one at a time. If we look at the yellow nodes, representing `Tx0`'s operations, we see no dependencies between them. We set no dependencies for nodes belonging to the same transaction, as the serial execution of a transaction itself guarantees no intra-transaction concurrent operations are ever performed. We only set the `prev` and `next` fields accordingly when appending a new operation node in such cases. After inserting both operations, the read's `next` field would point to the write node on the right, and the write's `prev` field would point to the read's node on its left.

The complexity arises when inserting a node in an already initialized linked-list containing operations from other transactions. Take for example `Tx1`'s first read from Figure 4.5. Since we already added a write from `Tx0`, the blue read will depend on it. This dependency is marked by the red arrow as an explicit dependency. In a more general sense, all `NonComm` reads must have an explicit dependency on the latest previous `NonComm` write if, and only if the `prev` field of that read operation points to an operation from a different transaction. We can also see this rule apply for the second read from `Tx2` (orange operation for SC 1, Key 2), as it has a dependency on the highest previous non-commutative write operation aimed for that same key.

Algorithm 4.1: Schedule Creation

```
Input: $BLOCK$ of transactions
Output: $QueuePartialReady$, $QueueReady$, $Schedule$, $Deps$, $DependentTxs$, $PartialReadyTxs$
1  $Schedule \leftarrow \emptyset$
2  $Deps \leftarrow \emptyset$                      // txId -> txs which it depends on
3  $DependentTxs \leftarrow \emptyset$              // txId -> txs dependent on txId
4  $PartialReadyTxs \leftarrow \emptyset$           // txId -> txs with first operation dependent on txId
5  $QueuePartialReady \leftarrow \emptyset$         // txs with first operation free of dependency
6  $QueueReady \leftarrow \emptyset$                // txs with all operations free of dependencies
7  foreach $Tx \in BLOCK$ do
8      $first\_operation \leftarrow true$
9      foreach $Op \in Tx$ do
10         if $Op$ is $Read$ then
11             HandleRead($Tx, Op, first\_operation$)      // Algorithm 4.2
12         else if $Op$ is $Write$ then
13             HandleWrite($Tx, Op, first\_operation$)     // Algorithm 4.3
14         $first\_operation \leftarrow false$
15     if $Deps[Tx] = \emptyset$ then
16         $QueueReady \leftarrow QueueReady \cup \{Tx\}$
17         $QueuePartialReady \leftarrow QueuePartialReady \setminus \{Tx\}$
18 return $QueuePartialReady, QueueReady, Schedule, Deps, DependentTxs, PartialReadyTxs$
```

Commutative reads represent somewhat an exception to this rule, since, due to their commutativity properties, such reads do not require reading the most recent value. These reads, by definition, are guaranteed to ensure correctness for the entire operation (which includes both the commutative read as well as the following commutative write) they are part of. As a result, these read operations can access the initial smart contract storage directly. During execution, the value read, along with the subsequent commutative write, is used to calculate the increment's delta. This delta is then stored in the `value` field of the node, as further explained in Section 4.6.

Writes, on the other hand, never conflict, as each has its own version, implicitly set to its transaction ID. An example of this is the write from `Tx4` aimed at `SC1,Key2`. As we can see from the illustration, there is no dependency set between that write node, and the previous operation (the orange node to the left). This guarantees correctness since that previous orange read will read from the blue write on its left, which can only be set by the thread executing `Tx1`. The schedule is then ready for execution.

Schedule Creation We now detail more formally the schedule creation process, with Algorithm 4.1.

As input, we expect the `BLOCK` of transactions containing, for each transaction, both the transaction's index in the block, as well as the ordered list of reads and writes. The `Schedule` structure will map smart contract's addresses to storage items, each mapping to a linked list of operations (this is what was represented in Figure 4.5). The `Deps` structure will map a transaction `Tx` to a set containing the transactions IDs transaction `Tx` depends on. `DependentTxs` will map a transaction ID to a set of all transactions depending on it. `PartialReadyTxs` will map a transaction ID to all other transactions that have their first operation depending on it. `QueuePartialReady` will contain all transaction IDs that have their first operation free of any dependency. This is because transactions need not wait for the dependent

Algorithm 4.2: Schedule Creation - Handle Reads

```
   Context: $QueuePartialReady$, $QueueReady$, $Schedule$, $Deps$, $DependentTxs$, $PartialReadyTxs$
1  procedure HandleRead($Tx$, $Op$, $first\_operation$):
2      $Schedule \leftarrow Schedule \cup Tx.ScAddress$
3      $Schedule[Tx.ScAddress] \leftarrow Schedule[Tx.ScAddress] \cup Op.item$
4      if $Op$ is $Commutative$ then
5          $Op.dependency \leftarrow \perp$
6      else
7          $latest\_non\_comm\_write \leftarrow$ Schedule.getLatestNonCommWrite($Op.item$)    // O(1)
8          $head \leftarrow \perp$    // stores head node if no latest_non_comm_write available
9          $node \leftarrow latest\_non\_comm\_write$
10         if $latest\_non\_comm\_write = \perp$ then
11             $Op.dependency \leftarrow \perp$
12             $head \leftarrow$ Schedule.getHead($Op.item$)    // O(1)
13             $node \leftarrow head$
14         $latest\_write \leftarrow \perp$
15         while $node \neq \perp \wedge node.Tx \neq Tx$ do
16             // Set dependencies for each write, be it commutative or not
17             if $node$ is $Write$ then
18                 $new\_entry \leftarrow DependentTxs[node.Tx] \cup \{Tx\}$
19                 $DependentTxs[node.Tx] \leftarrow new\_entry$    // Set Op as dependent on node
20                 $Deps[Tx] \leftarrow Deps[Tx] \cup \{node.Tx\}$    // Set node as a dependency of Op
21                 $latest\_write \leftarrow node$
22             $node \leftarrow node.next$
23         if $latest\_write \neq \perp$ then
24             if $latest\_non\_comm\_write \neq \perp \wedge latest\_non\_comm\_write.Tx \neq Op.Tx$ then
25                 $Op.dependency \leftarrow latest\_write$    // explicit dependency
26             if $first\_operation$ then
27                 // Set Tx as partial ready transaction for the last write
28                 $PartialReadyTxs[latest\_non\_comm\_write.Tx] \leftarrow Tx$
29         else if $first\_operation$ then
30             $QueuePartialReady \leftarrow QueuePartialReady \cup \{Tx\}$
31     $Schedule[Tx.ScAddress][Op.item] \leftarrow Schedule[Tx.ScAddress][Op.item] \cup Op$
```

transaction to finish executing, but rather for the specific dependent operation of that transaction to finish executing. So a transaction in `QueuePartialReady` will be able to start executing, but with the possibility of needing to wait upon some operation, as only the first operation of this transaction is free of dependencies. This provides a finer concurrency control granularity, allowing transactions to execute as soon as possible. `QueueReady` will hold all transaction IDs that have no dependencies. As explained in the example earlier, we start by running over each transaction in the block, and over each operation of the current transaction (lines 7 and 9).

We will explore first the case of read operations (Algorithm 4.1 lines 10-11) described by Algorithm 4.2. If the operation is a commutative read, then we mark no dependency (lines 4-5), meaning this operation will read from the initial storage at runtime. This is because, as explained earlier, these reads do not require reading the most recent value. If the read is non-commutative, then we fetch the most recent non-commutative write up to the current operation being inserted (line 7). This operation is $O(1)$, as the last write nodes are cached during schedule building. If no such write exists, then we fetch the head of the chain of operations for that state item, and set it as the starting node (lines 10-13). The goal here is to

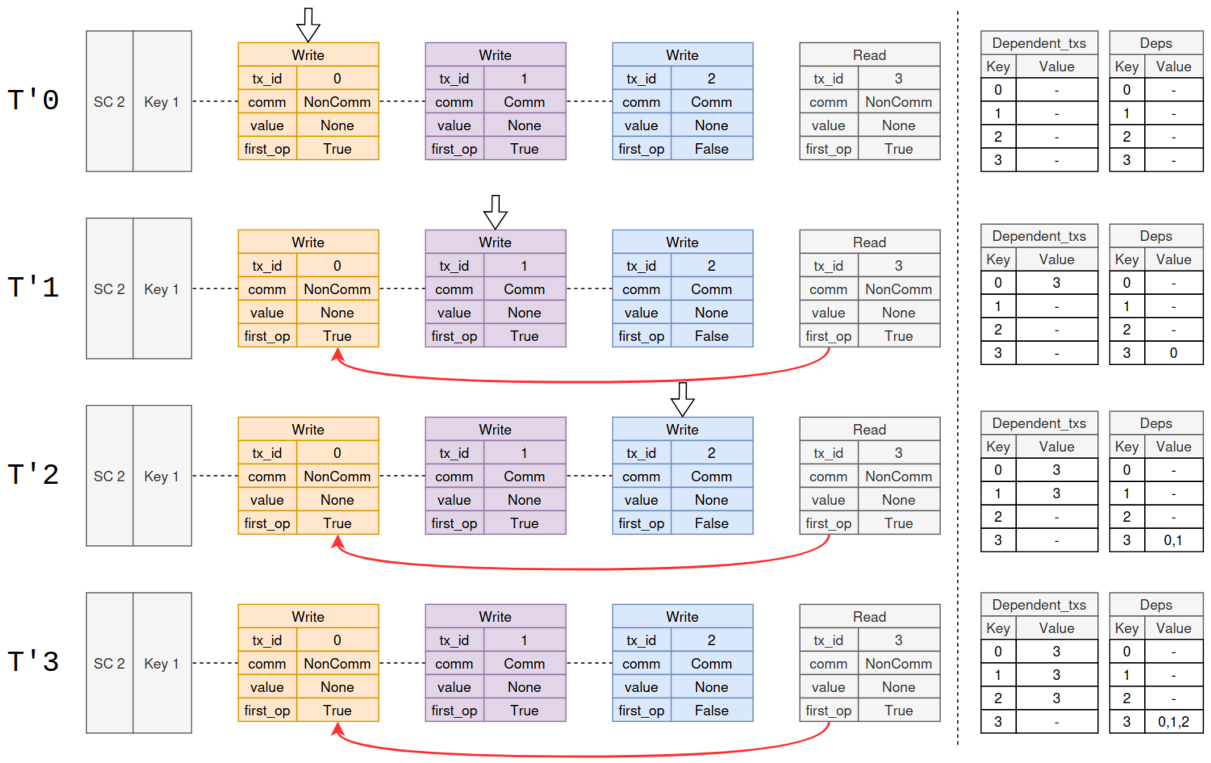


Figure 4.6: Schedule Creation - Dependency setting for a non commutative read.

iterate all nodes to the right, starting from the latest non commutative write, or from the start of the linked list if no non commutative write is available (lines 15-22). For each iteration, we are only concerned with write operations (line 17) as these are the only type of operation that conflicts with a non-commutative read. Upon matching one, we update the dependencies both for the non-commutative read's transaction and for the write's transaction. Note that, since we start iterating on the last non-commutative write (if any), that, by definition, is the latest one, then only the first write of this iteration is non-commutative, or none is. All following writes will always be commutative.

After iterating, if we did find any write before our non-commutative read (line 23), then it could be either a non-commutative write, or a commutative one, in case we used the head of the list as a starting point. Thus, we check if we have the `latest_non_comm_write` set, and if that write is from another transaction (line 24). If `true`, we set it as an **explicit dependency** for the current non commutative read being parsed (line 25). Recall that we do not set dependencies between operations from the same transaction. If the `latest_write` is defined and we are parsing the first operation of the current transaction , then we set the current transaction as a partial ready for `latest_write` (lines 26-28).

If there is no previous write and it is the first operation, then we insert it into the `QueuePartialReady` set. This is possibly a temporary insertion, since if later on, if all operations from this transaction are free of dependencies, we remove it from the `QueuePartialReady`, and insert it into the `QueueReady`.

Algorithm 4.3: Schedule Creation - Handle Writes

```
   Context: $QueuePartialReady$, $QueueReady$, $Schedule$, $Deps$, $DependentTxs$, $PartialReadyTxs$
1  procedure HandleWrite($Tx$, $Op$, $Schedule$, $Deps$, $PartialReady$, $first\_operation$):
2      $Schedule \leftarrow Schedule \cup Tx.ScAddress$
3      $Schedule[Tx.ScAddress] \leftarrow Schedule[Tx.ScAddress] \cup Op.item$
4      $Op.dependency \leftarrow \bot$
5      $Op.first\_operation \leftarrow first\_operation$
6      if $first\_operation$ then
7          $QueuePartialReady \leftarrow QueuePartialReady \cup \{Tx\}$
8      if $Op$ not $Commutative$ then
9          Schedule.setLastNonCommWrite($Op$)
10     $Schedule[Tx.ScAddress][Op.item] \leftarrow Schedule[Tx.ScAddress][Op.item] \cup Op$
```

Figure 4.6 shows an example on how the schedule structures are updated with the insertion of a new non-commutative read illustrated by the grey node on the right. The tables on the right side represent the `Deps` and `DependentTxs` present in the previous algorithms. At timestamp `T'0` we fetch the latest non-commutative write. In this example there was one (the orange node), so we set it as the dependency, illustrated by a red arrow. Since the transaction ID of the node being inserted is 3, we place 3 in the key 0 of `DependentTxs` to specify the orange transaction (transaction 0) has a dependent transaction on it. We also place 0 in the key 3 of `Deps` to specify the inverse. Note that, in the image, these updates are only visible at the start of the next timestamp - `T'1`. Then we go to the next node, and do the exact same thing, but without setting any explicit dependency, since now the (purple) node represents a commutative operation. The same goes for the blue node. At `T'3` we can see all the dependencies are marked.

4.5.1 Eliminating Write-Write Conflicts

For write operations (Algorithm 4.1 lines 12-13), the procedure is much simpler, as shown in Algorithm 4.3. When a transaction writes to a particular data item, it does not overwrite previous values. Instead, it creates a new version of the data, marked by its `tx_id`. This allows multiple versions of the same data to coexist, with each transaction effectively writing to its own isolated version. Consequently, writes from transactions that follow in the block do not need to depend on earlier writes, as they will be writing to their own separate version. As such, write nodes never have dependencies.

Similarly, **read-write conflicts**, where one transaction reads data while another attempts to modify it, are also resolved through write versioning. Since the system respects the transaction ordering of the block, reads followed by writes from different transactions no longer result in conflicts. This is because each write operation is applied to its own version of the data, ensuring that the read operation remains unaffected by subsequent modifications. Specifically, a write that occurs after a read will always have a higher version number than the one the read operation accesses, preventing any interference.

Figure 4.7 illustrates this scenario, where multiple transactions issue operations, with a read-write conflict indicated by the red arrow. The arrows represent dependencies, and the dashed line shows the

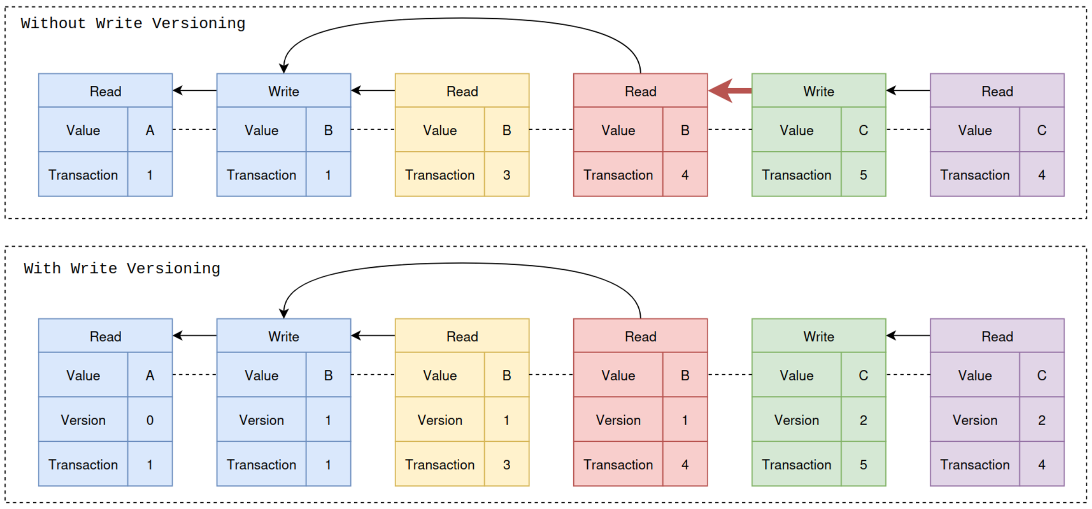


Figure 4.7: Write Versioning.

sequence of operations following the block order. With write versioning enabled, the green write operation no longer conflicts with the red read, as the read accesses a lower version. As a result, transaction 5 can execute immediately, without waiting for the preceding three transactions to complete, significantly increasing parallelization.

Going back to the algorithm, if it is the first operation of the transaction, we insert in `QueuePartialRead` (lines 6-7). Finally, if the write is non commutative, then we update the last non commutative write in the `Schedule` (lines 8-9).

4.5.2 Schedule Parallelization

Having discussed the sequential scheduling algorithm, we now shift our focus to the parallelized version, which is essential for ensuring efficiency with larger blocks.

The parallelized scheduling algorithm builds upon the serial version, but introduces a **merging mechanism** to combine schedules from multiple threads. The idea is to divide the block into N **subsets**, where N represents the number of available threads. Each subset contains transactions that are sequential in the block's order; for instance, thread 0 processes transactions 0 through $N-1$, thread 1 processes transactions N through $2N-1$, and so forth.

This strategy allows each thread to maintain a partial view of the block. By applying the serial scheduling algorithm locally, each thread builds a **partial schedule** for its subset. Importantly, since transactions within each subset respect block order, merging two partial schedules only requires updating the dependencies at the frontier between the two schedules.

Once each thread has completed constructing its partial schedule, a **binary tree merging strategy** is employed to combine these schedules. In this process, threads are paired, and their schedules are

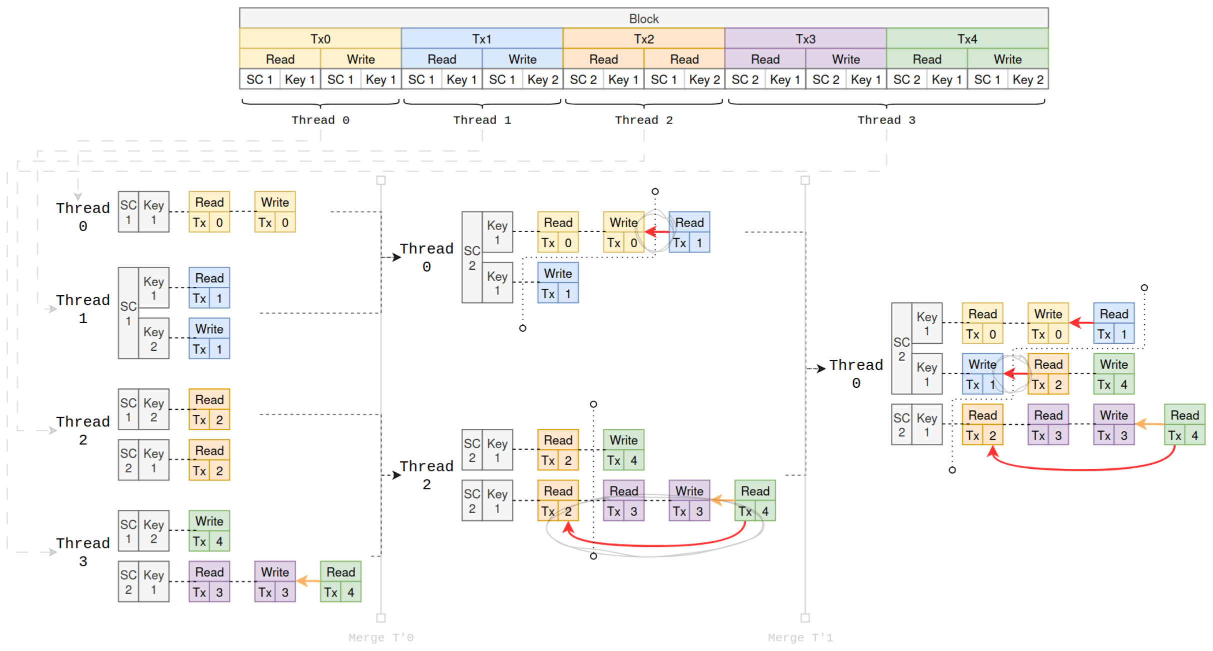


Figure 4.8: Parallel schedule creation.

merged recursively in a pairwise manner. With each merge step, the number of schedules is halved, reducing the total number of required merges at each level. This process continues until all schedules are merged into a single final result, completing in $\log(N)$ rounds. The key advantage of this approach is that it limits the number of merge operations, which keeps the total merging overhead manageable.

Running Example Figure 4.8 illustrates the parallelized schedule creation process. At the top, we observe the same block used in previous examples. Initially, the block is divided among the available threads, where each thread sequentially constructs its own partial schedule. At `T'0`, as represented by the vertical lines terminating in squares, the first merge occurs: `Thread 0` merges with `Thread 1`, while `Thread 2` merges with `Thread 3`. Following this initial merge, `Thread 1` and `Thread 3` complete their execution. The merging process may introduce new dependencies between the partial schedules at the merge frontier, which is marked by the dotted line ending in circles—separating the schedule with lower transaction IDs (on the left) from the schedule with higher transaction IDs (on the right).

After `Merge T'0`, we can observe that `Thread 0` introduces a new dependency, represented by the circled red arrow. The same color scheme from Figure 4.5 is maintained: red arrows indicate explicit dependencies, while orange arrows denote implicit dependencies.

Similarly, during the merge between `Thread 2` and `Thread 3`, a new dependency emerges between transaction 4 and transaction 2. Note that this dependency is not strictly positioned at the frontier with respect to the right-side schedule. Dependencies are not necessarily drawn between the first transaction of the right-side schedule and the last transaction of the left-side schedule. Instead, they form between

Algorithm 4.4: Parallel Schedule Creation - Binary Tree Merging

```
   Input: $transactions$ - block of transactions
 1 procedure BuildScheduleFromTransactions($transactions, n\_threads$):
 2     if $\#transactions = 0$ or $n\_threads = 0$ then
 3         return
 4     $n\_threads \leftarrow$ min($\#transactions, n\_threads$)
 5     $tree\_levels \leftarrow \lceil \log_2(n\_threads) \rceil$
 6     $txs\_per\_thread \leftarrow \lfloor \#transactions / n\_threads \rfloor$
 7     $deps \leftarrow$ InitDeps($n\_threads$)
 8     $thread\_states \leftarrow$ InitStates($n\_threads$)
 9     for $id \leftarrow 0$ to $n\_threads - 1$ do
10         $deps[id] \leftarrow \langle\ \rangle$   // empty queue
11         // Get the thread to which 'id' should send to.  Each thread only sends data once.
12         $receiver\_id \leftarrow$ GetReceiverThread ($id$, $tree\_levels$)
13         if $receiver\_id \neq \perp$ then
14             $deps[receiver\_id]$.push($id$)
15     for $id \leftarrow 0$ to $num\_threads - 1$ do
16         $txs\_subset \leftarrow$ GetSubsetOfTxs($transactions, txs\_per\_thread$)
17         ExecuteThread($id, thread\_states, txs\_subset, dependencies[id]$)
18     // Thread 0 will contain final merged state
19     WaitForCompletion ($thread\_states[0]$)
20     return $thread\_states[0]$
21 procedure ExecuteThread($thread\_id$, $thread\_states$, $txs\_subset$, $dependency\_list$):
22     $schedule \leftarrow$ BuildSchedule($txs\_subset$) // serial schedule
23     while $dependency\_list \neq \perp$ do
24         $dep\_id \leftarrow dependency\_list$.pop()
25         Wait ($thread\_states[dep\_id]$)
26         $schedule$.merge($thread\_states[dep\_id].schedule$)
27     $thread\_states[thread\_id] \leftarrow schedule$
28     FinishThread ()
29 procedure GetReceiverThread($thread\_id$, $tree\_levels$):
30     for $level \leftarrow 0$ to $tree\_levels - 1$ do
31         $step \leftarrow 2^{level}$
32         if $\lfloor thread\_id / step \rfloor \% 2 \neq 0$ then
33             return $thread\_id - step$
34     return $\perp$
```

the **first non-commutative reads** from the right-side schedule and both the **last non-commutative writes** and **last commutative writes** from the left-side schedule, as we will explain later.

Finally, `Thread 0` and `Thread 2` merge at `T'1`, introducing another dependency, this time at `SC1,Key2`, from transaction 2 to transaction 1. The resulting schedule is identical to the one produced by the sequential approach, as shown in Figure 4.5.

Binary Tree Parallel Execution Algorithm 4.4 describes the parallel binary tree merging. The number of threads is set to the minimum between the total transactions in the block, and the number of desired threads (line 4) as it does not make sense to have more threads than transactions. Next we define the number of levels for the binary tree as the ceiling of the logarithm base 2 of the total number of threads. This defines the upper bound for the total leaf nodes covered by the tree with $\lceil \log_2(n\_threads) \rceil$ number of levels. We also get the number of transactions per thread, taking into account uneven distribution.

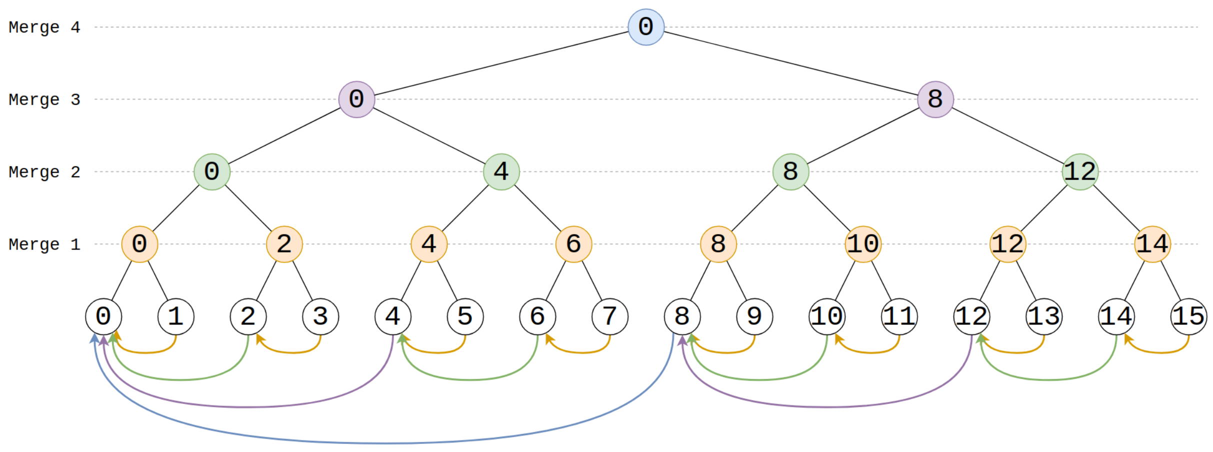


Figure 4.9: Binary merging tree. Arrows represent to which thread some thread has to send its schedule to. Colors represent the merging step in which the sending thread will send its schedule to the receiving thread.

We could take the simple approach, and assign $\lfloor \#transactions/n\_threads \rfloor$ to each thread, leaving the rest to the last thread. But this becomes problematic as the thread number increases. As an example, assuming all transactions take the same time to execute, if we set $n\_threads$ to 64, and assume 1000 transactions, then each thread would be assigned $\lfloor 1000/64 \rfloor = 15$ transactions, and the last thread would be assigned $15 + (1000 - 15 * 64) = 55$ transactions. This means in practice that the last thread would delay the entire parallel execution by $55/15 \approx 3.67$ times! Instead, we assign the first threads one more transaction than the 15 until the total remaining transactions is multiple of 15. This way the longest execution will be only around $16/15 \approx 1.07$ times longer, which is much more acceptable.

We rely on two basic structures: `deps` (line 7) represents a mapping from some thread A to the threads it has to wait upon, in the correct order. Each thread only sends its local schedule at most once to another thread, and only even threads receive a schedule from other threads, but the receiving threads can receive a schedule from more than one thread; `thread_states` (line 8) is also a mapping from a thread ID to its partial schedule. This is used to share a thread's state with another thread for the merging operation.

We then start by running over each thread A, initializing the queue of transactions that thread A will need to wait upon (line 10). We then compute, for thread ID, to which thread this needs to send to. This is done by the `GetReceiverThread` procedure, which takes into account all levels. As an example, consider thread 12 in a system with a total of 16 threads, organized into a binary tree with 4 levels. To determine which thread, thread 12 must send its schedule to, then the only x in $\{0, 1, 2, 3\}$ (line 30) such that $\lfloor 12/2^x \rfloor$ results in an odd number (lines 31-32) is $x = 2$, translating to $step = 2^2 = 4$ (line 31). Thus, we get that thread $12 - 4 = 8$ (line 33) is the thread to which thread 12 will need to send its schedule to. This will only happen at the third merging step, but is marked before starting the execution. Figure 4.9 illustrates the messages that need to be sent between any two threads, represented by arrows, each with the same color of the merge step in which they will happen - orange refers to merging step 1, green to merging step 2, purple to merging step 3 and blue to merging step 4. As we can see, thread 12 will only send a message to thread 8, at merging step 3 (purple color).

Since we call `GetReceiverThread` in sequential thread order (line 12, Algorithm 4.4), the `sender_thread` IDs for each receiver thread are correctly set in this sequence. For example, as illustrated in Figure 4.9, the receiver thread 0 will have its first sender as thread 1 (orange arrow: $1 \rightarrow 0$), followed by thread 2 (green arrow: $2 \rightarrow 0$), thread 4 (purple arrow: $4 \rightarrow 0$), and finally thread 8 (blue arrow: $8 \rightarrow 0$). Thus, the `deps` entry for thread 0 will be $\langle 1, 2, 4, 8 \rangle$.

After setting the `deps` for each thread (in practice, only even threads will have an entry), we assign a subset of the block to each thread as described previously, and begin building the partial schedule over that subset (lines 15-17).

The work performed by each thread, encapsulated in the `ExecuteThread` procedure (lines 21-28), is conceptually simple except for the merging step. Each thread first constructs its partial schedule following Algorithm 4.1 from its assigned subset of transactions, then waits for the schedules from other threads (line 25) it is designated to receive from. It follows the correct merging order and combines the received schedule with its own. When a thread no longer needs to wait for any other schedules (i.e., it exits the while loop at line 23), it writes its completed schedule to `thread_states` (line 27) so that the next dependent thread (waiting at line 25) can access it. The thread then signals that its execution is complete (line 28).

Meanwhile, the main thread waits for thread 0 to finish merging all schedules (line 19). It is sufficient to wait only for thread 0, as this thread is guaranteed to hold the final merged schedule. Once thread 0 completes, the resulting schedule is returned.

Schedule Merging We will now take a closer look at how the merging process itself is done. The definition for dependencies is still the same as before, meaning there will be only dependencies from non-commutative reads on both commutative and non-commutative writes preceding it. Since Algorithm 4.1 captures all dependencies, both schedules being merged are complete with respect to dependency tracking. Meaning the only work to be done at merging, is to try and find possible new dependencies between both schedules.

We shall refer to **self schedule** as the schedule from the thread receiving the message, and to **other schedule** as the schedule being sent by the sending thread (note that schedules are not really being sent in the sense of message passing, but rather written into memory). One interesting property is that since transactions in the block are ordered, and we assign a subset to a thread following the threads IDs in increasing order, threads with higher IDs get a sequence of contiguous transactions with higher IDs than the ones from threads with a lower IDs. This allows us to only have to look for dependencies from

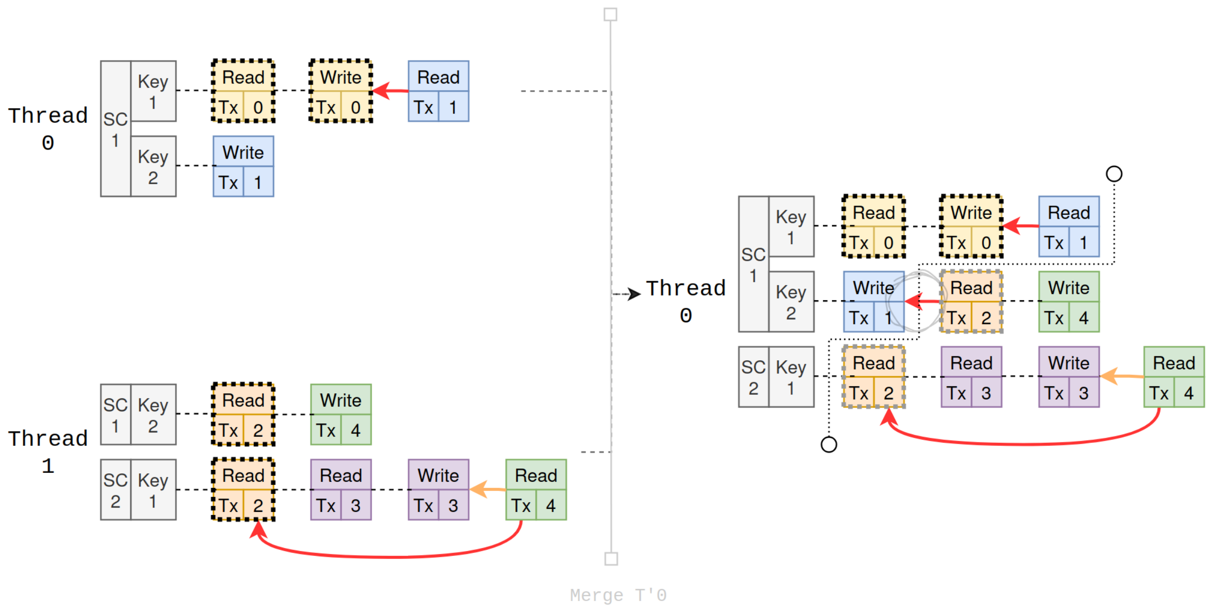


Figure 4.10: Schedule Merging. Black dashed boxes identify operations from a transaction placed in `QueueReady`. Grey dashed boxes identify operations from transactions in `QueuePartialReady`. All other representations have the same meaning as in Figure 4.5

the *other* schedule to the *self* schedule. This is also enforced by the fact that we respect transaction ordering in the merging steps, i.e., for any two transactions' merge, there is no other transaction that has a schedule with transactions in between the ones both of the two merging transactions have. As before, we will run first through an example, and then we discuss the algorithm.

Figure 4.10 illustrates the merging steps. This figure depicts the exact same merging step `T'1` from Figure 4.8, but now in more detail. Black dashed borders represent operations from transactions in the `QueueReady`, while grey dashed borders represent operations from transactions in the `QueuePartialReady`. All other representations have the same meaning as in Figure 4.8. As a starting point, we assume both schedules from thread 0 and 1 have finished building. As we can see from `Thread 0`'s schedule, `Tx0` is placed in the `QueueReady` since it has no dependencies, and `Tx1` is not in any queue since neither its first operation (blue read) has no dependencies, nor all operations are free. As for `Thread 1`'s schedule, only `Tx2` is free of any dependencies, thus it is placed in `QueueReady`.

We will now go over each SC key at a time, trying to find possible new dependencies from the merging at the frontier between the two schedules. Starting at `SC1,Key1`, we can see that `Thread 1`'s schedule has no entry for that key, thus the resulting schedule from the merging at that item will just be the original one from `Thread 0` without any alteration. Next, for `SC1,Key2` we can see that `Thread 1`'s schedule has two operations from transactions with higher IDs. Thus, we will append these operations (orange and green) at the end of `Thread 0`'s `SC1,Key2` entry. We now search for the first non commutative read from the right schedule. In this case, it is the orange read from `Tx2`. Then we will find the last write from the

Algorithm 4.5: Parallel Schedule Creation - Merging

```
   Input: $self$ - current schedule, $other$ - schedule to be merged
   Context: $starting\_tx\_id$ - ID of the first tx in the $self$ schedule
1  procedure MergeSchedules($self$, $other$):
2      // Extend vectors for dependencies, transactions, and other relevant data
3      $self.Deps \leftarrow self.Deps \cup other.Deps$
4      $self.DependentTxs \leftarrow self.DependentTxs \cup other.DependentTxs$
5      $self.PartialReadyTxs \leftarrow self.PartialReadyTxs \cup other.PartialReadyTxs$
6      Merge($self$, $other$)
7      // Merge the queues after having updated the txs from the 'other' schedule's queues
8      $self.QueueReady \leftarrow self.QueueReady \cup other.QueueReady$
9      $self.QueuePartialReady \leftarrow self.QueuePartialReady \cup other.QueuePartialReady$
10 procedure Merge($self$, $other$):
11     foreach $sc\_other \in other$ do
12         foreach $key\_other \in sc\_other$ do
13             $linked\_list\_other \leftarrow other.GetLinkedList(sc\_other, key\_other)$
14             $linked\_list\_self \leftarrow self.GetLinkedList(sc\_other, key\_other)$
15             $last\_write\_self \leftarrow self.GetLastWrites(sc, key\_other)$
16             if $linked\_list\_self$ exists then
17                 // Joins linked lists by setting the .next and .prev fields of the frontier nodes, and also updates
                   the .tail of self linked list.
18                 join_linked_lists($linked\_list\_self$, $linked\_list\_other$)
19                 if $last\_write\_self \neq \bot$ then
20                     $first\_non\_comm\_read\_other \leftarrow other.GetFirstNonCommRead(sc\_other, key\_other)$
21                     $SetFrontierDependency(last\_writes\_self.non\_commutative, node\_ref)$
22                     if $last\_write\_self$ is $NonCommutative$ then
23                         $node.dependency \leftarrow last\_writes\_self.non\_commutative$
24             else
25                 // If no operations in self, initialize the list with other's operations
26                 $self.Schedule[sc\_address][key] \leftarrow self.Schedule[sc\_address][key] \cup linked\_list\_other$
```

self schedule. Since this write is non-commutative (we assume the same operation commutativity as in Figure 4.5), then we have to mark a **new dependency** between the orange and blue operations as circled in Figure 4.10. Also, since Tx2 was in QueueReady before the merge, and now has a new dependency on a node that is not the first, we take it out of QueueReady, and push it to QueuePartialReady, as it can still have its first operation free of conflicts. Finally, for SC2,Key1, we see that Thread 0's schedule had no entry for that key, so Tx2's first operation remains without dependencies, meaning it will remain in QueuePartialReady. The merging is finished by now, and we end up with Tx0 in QueueReady, and Tx2 in QueuePartialReady, as well as a new dependency from Tx2 to Tx1.

Algorithm 4.5 describes the steps in detail. As inputs we receive both the *self* and *other* schedules. We start by extending all the dependencies marked in *self* with the ones from *other* (lines 3-5). This is true for both Deps and DependentTxs, which, as we described earlier, track all the **implicit dependencies** between any two nodes in both directions. We also merge the PartialReadyTxs, the mapping of a transaction to all other transactions whose first operation only depends on it. We then do the schedule per-key merging, as expressed by the Merge procedure in lines 10-26.

We iterate over keys of all smart contract addresses in the right schedule, and fetch the linked list holding the chain of operations to that key from both schedules (lines 13-14). If the *self* schedule does

Algorithm 4.6: Parallel Schedule Creation - Dependency Update During Merging

```
   Input: $self$ - current schedule, $other$ - schedule to be merged
   Context: $starting\_tx\_id$ - ID of the first tx in the $self$ schedule
1  procedure SetFrontierDependency($node\_self, node\_other$):
2      $self.Deps[node\_other.tx] \leftarrow self.Deps[node\_other.tx] \cup node\_self.Tx$
3      $self.DependentTxs[node\_self.tx] \leftarrow self.DependentTxs[node\_self.tx] \cup node\_other.Tx$
4      $ready\_tx \leftarrow \perp$
5      if $node\_other.Tx \in other.QueueReady$ then
6          $ready\_tx \leftarrow node\_other$
7          $other.QueueReady \leftarrow other.QueueReady \setminus node\_other.Tx$
8      $node \leftarrow node\_self$
9      while $node \neq \perp \land node.tx \neq node\_other.Tx$ do
10         if $node$ is $Write$ then
11             $self.Deps[node\_other.tx] \leftarrow self.Deps[node\_other.tx] \cup node\_self.Tx$
12             $self.DependentTxs[node\_self] \leftarrow self.DependentTxs[node\_self] \cup node\_other$
13         $node \leftarrow node.next$
14     if $node\_other$ is $FirstOperation$ then
15         $other.QueuePartialReady \leftarrow other.QueuePartialReady \setminus node\_other.Tx$
16         $self.PartialReadyTxs[node\_self.Tx] \leftarrow self.PartialReadyTxs[node\_self.Tx] \cup node\_other.Tx$
17     else if $ready\_tx \neq \perp$ then
18         $self.QueuePartialReady \leftarrow self.QueuePartialReady \cup ready\_tx$
```

not have such entry, we just append the right's entry, and the job is done for that key (line 26). Else, if the *self* schedule has an entry for that key, then we join both linked lists together (line 18), appending the *other*'s one at the end of the *self*'s. This is done by setting the *self*'s tail `next` pointer to the *other*'s head, and the *other*'s head `prev` pointer to *self*'s tail. We also update *self*'s `tail` pointer to *other*'s tail node.

We then proceed to finding new dependencies. We fetch the last write from the *self* schedule (line 15), and if there is such write, we fetch the first non commutative read from the right schedule (line 20). Since the last write operation can be commutative or non-commutative, and we only set the `dependency` field in a node if it captures an **explicit dependency**, i.e, a dependency from a non-commutative read to a non-commutative write, we need to check this before setting that field (lines 22-23).

Having the *self*'s last write and the *other*'s first non commutative read we then update all possible dependencies (line 21, Algorithm 4.5) as described in Algorithm 4.6. We start by setting the implicit dependencies both ways (lines 2-3), and pop the non commutative read if it was in `QueueReady` (lines 5-7). This is because a transaction ready to be executed cannot have any dependency. Since we are setting a new dependency, that transaction can no longer be ready. It can, although still be partial ready, if its first operation still remains free of conflicts, as we will see in a moment.

We then iterate over all nodes from the *self*'s last write to the end of the list (lines 9-12). The rational is that since `node_self`, i.e., the starting node is either the last non-commutative write, or a commutative write, then by definition, there can be never a non-commutative write afterwards. Only reads and commutative writes. Since commutative writes still incur dependencies with non-commutative reads, we have to find all of such writes that come afterwards the last non-commutative write and mark them as dependencies (lines 11-12). Care must also be taken not to go over the `other` transaction id (line 9) as the linked lists have already been merged. Finally we need to check if this was the first operation of `other`'s transaction. If so, since it now has at least one new dependency, it is inserted into that dependencie's `PartialReadyTxs` set, and removed from the `QueuePartialRead` (lines 14-16). Finally, if the transaction responsible for the non-commutative read of the right schedule was previously marked as ready, and this non-commutative read is not its first operation, then this means its first operation is still free of conflicts, and we push it to `QueuePartialReady` (lines 17-18).

After merging all schedules, the resulting schedule will need to be converted to a **concurrent schedule** which introduces some concurrency control features. Namely, for both `QueueReady` and `QueuePartialReady`, we need to ensure correctness when pushing and popping from both queues, and we also need to introduce a vector that keeps track of the execution state of each transaction. Each transaction can be in one of three states:

- **NotExecuted:** Default state. All transactions start with this state set.
- **Executing:** Set for transactions that are currently executing.
- **Executed:** Set for transactions that have already finished execution.

We will see in the next section when and how exactly these states change over execution. Besides this state tracking, we also improve the `Deps` map. Instead of just storing the set of transactions that are dependencies of the transaction whose ID is the key for that entry, we now also store an atomic counter initialized with the size of the set. This will become useful to track transaction's dependencies at runtime.

The schedule is ready to be fed to the next step: Schedule Execution.

4.6 Schedule Execution

Upon constructing the execution schedule with all dependencies tracked, the system is prepared to initiate the transaction execution phase. Concurrency control is implemented at two distinct levels: the **transaction selection level** (queue level) and the **operation execution level** (operation level). These mechanisms ensure the orderly and conflict-free execution of transactions in a parallel environment.

At the queue level, two specialized structures manage the transaction execution readiness:

- `QueueReady`: This queue holds transactions that are entirely free of dependencies, meaning they can immediately start executing without waiting for any prior operations to complete.
- `QueuePartialReady`: This queue contains transactions whose first operation is independent of other transactions, allowing them to start execution. However, these transactions may need to suspend execution at some later point to wait for specific operations from other transactions to complete.

The queue-based concurrency control governs the initiation of transaction execution. It ensures that only transactions that are either fully independent - QueueReady - or partially independent - QueuePartialReady - are eligible for execution. This level of control focuses on determining which transactions can be selected for execution at any given time based on their dependency status.

In contrast, once a transaction has been dequeued and its execution has started, a second level of concurrency control occurs at the operation level. At this stage, individual operations within the transaction are subject to locks, ensuring that non-commutative operations do not conflict with concurrent transactions. Specifically, non-commutative read operations must wait until the corresponding write operations have fully completed, ensuring data consistency. For commutative operations, the concurrency control becomes more nuanced, as we will discuss in subsequent sections. Nevertheless, the core principle remains the same: **non-commutative read operations must wait for the associated write operations to finish** before proceeding with execution.

It is important to note that this operation-level concurrency control is only relevant for transactions from QueuePartialReady. Transactions from QueueReady, which have no dependencies, can execute without delay, as their operations are free of conflicts. This allows us to be more flexible in their execution - the order in which these transactions are pushed into the queue does not need to be respected during execution meaning they can be popped in any order. This introduces the possibility for mechanisms that allow parallel popping of transactions from QueueReady, further enhancing execution throughput by enabling multiple transactions to be processed concurrently without dependency conflicts.

Running Example We will first go through a practical example to clarify the workflow, and then we delve into the algorithm details. We will refer to Figure 4.11 throughout this example. The top half represents the initial schedule. As before, we use black dashed borders around the first node of ready transactions, and a grey dashed border around the first operation of partial ready transactions. We have also included the field op_idx on each node to represent the order of the operation within that transaction, just to make clear the order the operations should be executed within a transaction. This is field has no practical use besides helping with the explanation. The bottom half represents a timeline of execution marked via dashed vertical lines with timestamps T'0, T'1 and so on, for ease of reference. We assume three threads for this example.

For this schedule, both Tx0 (yellow) and Tx3 (purple) are free of conflicts, so they are placed in QueueReady. As for Tx1 (blue) and Tx4 (green), since their first operation is free of conflicts, but still have other operations with conflicts, they are placed in QueuePartialReady. As we can see, at T'0, before starting executing, QueueReady is initialized with $< 0, 3 >$ and QueuePartialReady with $< 1, 4 >$. We also represent the Executed counter that tracks the number of executed transactions, which is used for the termination condition.

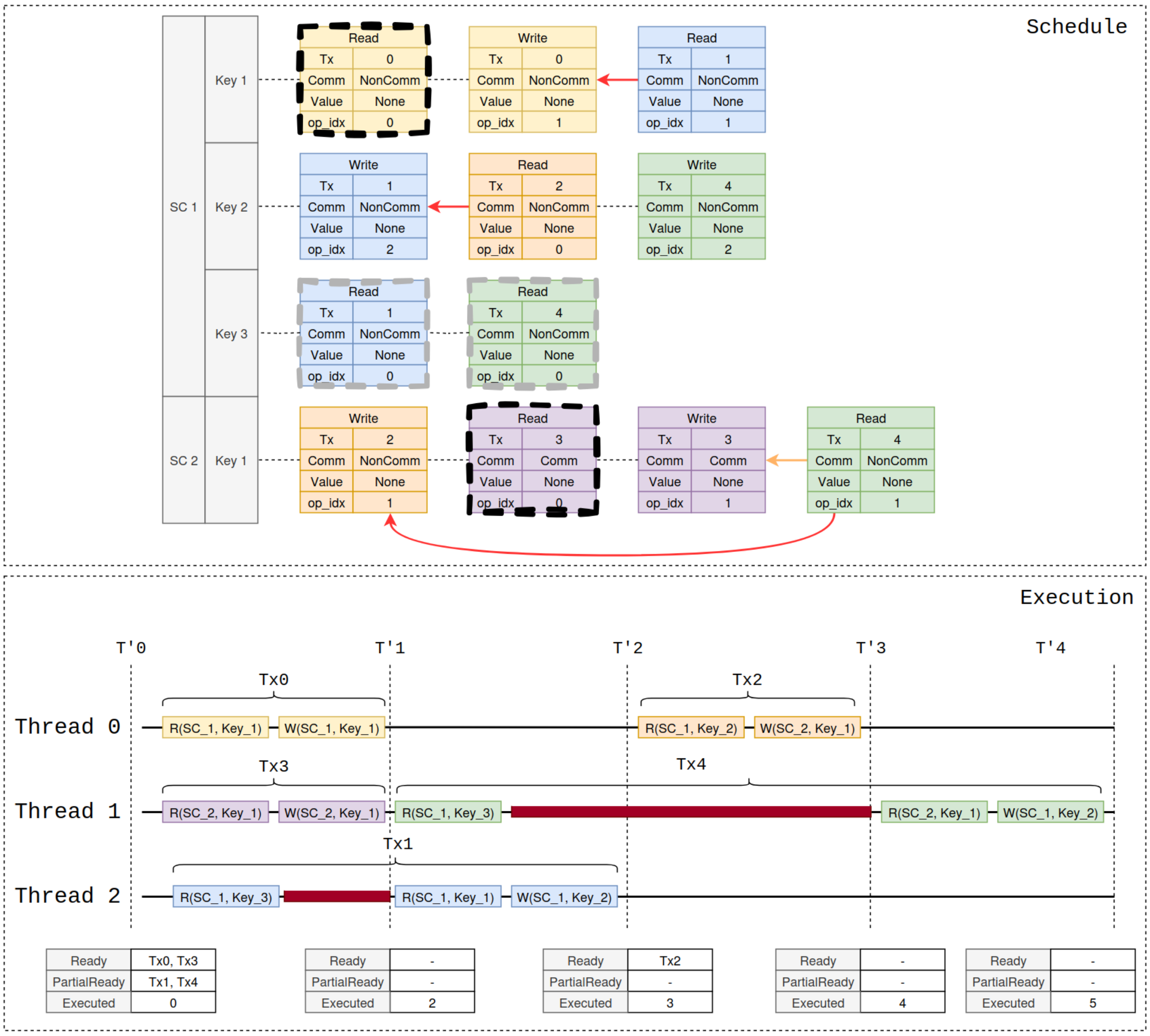


Figure 4.11: Schedule Execution. Red lines in the execution diagram represent waiting for some operation to complete.

The first step of execution is for all threads to try popping a transaction from `QueueReady`. As we can see, `Thread 0` and `Thread 1` popped all available transactions from that queue. In such cases, when `QueueReady` is empty, threads try popping from `QueuePartialReady`, which is the case for `Thread 2`. At `T'1` both `Thread 0` and `Thread 1` have finished executing their transaction. When this happens, each thread will go though its `DependentTxs` map and fetch all transactions that depend on the finished transaction. Taking `Tx0` as an example, we can see from the schedule in Figure 4.11 that there is a read from `Tx1` aimed at `SC 1, Key 1` that depends on a write from `Tx0`. Thus `Thread 0` will decrease the counter of dependencies for `Tx1`. Since that was the only dependency, the transaction could be pushed to `QueueReady`. But since `Thread 2` is already executing it, `Tx1`'s state has already been changed to `Executing`, meaning we do not push it to any queue. `Thread 1` does the same for its finished transaction Tx3 - finds dependencies, and decreases the dependency counter. In this case it decreases the dependency counter of Tx4. Since Tx4 has two dependencies, decreasing one will leave it still with one. So we cannot push it to the QueueReady. Besides decrementing the dependency counter, there can sometimes be cases where the first operation of a dependent transaction depends on the transaction finished executing. In these cases the dependending transaction is pushed to the QueuePartialReady instead. If, later on, we end up freeing all of its dependencies, then we move it from QueuePartialReady to QueueReady.

Referring back to our example, besides Thread 0 and Thread 1, we can also see that Thread 2 did not finish executing at T'1. We use dark red to represent waiting states. After finishing the first read of SC1, Key3, we will go to the second read of Tx1, which is represented as the blue operation aimed at SC1, Key1. As we can see this operation depends on the yellow write from Tx0, and as such we need to wait until that write is finished. Because of this, Thread 2 needs to stall execution until T'1, which is when Thread 0 finished that write operation from Tx0. Only then Thread 2 resumes execution, and can read the written value. Still in T'1, since both Thread 0 and Thread 1 finished executing their transactions, each thread will try to pop a new transaction. Since only Tx4 was remaining from T'0, one of the threads picks it up and starts executing it. Again, since Tx4 depends on the last write of Tx2, as represented by the red arrown in SC2, Key2, we will need to wait until that write has finished.

At the end of T'1 we end up with one more transaction finished (Tx1), and we increment the Executed count to 3. Since Tx1 finished executing, we look up again for its dependencies, and find that Tx2 depends on it with that dependency being its only one. So as we decrease it, we also push it to the QueueReady.

At T'2, since both Thread 0 and Thread 2 have finished executing, one of them pops Tx2 and executes it. Notice that Thread 1 is still waiting on the write of Tx2. Only at T'3, when Tx2 has finished writing, we can read the value written. Finally, at T'4, when Thread 1 finished executing Tx4, it increases the Executed counter to 5 and notifies all waiting threads. Each will then verify that the Executed counter equals the initial number of transactions, and execution is finished.

Schedule Execution We will now go through the algorithm, specifying more formally the steps involved. The execution phase (Algorithm 4.7) starts by iterating over all available threads and assign them work to do (lines 2-8). Each thread loops until all transactions have been executed. For each iteration it tries fetching a transaction ID, either from the QueueReady or from QueuePartialReady (line 4). If no transaction ID is returned, then all transactions were executed, and the execution phase is ended. Otherwise, the thread fetches the transaction of that ID from the block (line 6) and starts executing the transaction (line 7). The ExecuteTransaction function will, among other things, choose a CVM from the ones available, and delegate the execution to the WVM, as explained in Section 4.1, which then invokes db_read and db_write methods to the CVM, which we use to synchronize execution. Thus, we

Algorithm 4.7: Schedule Execution

```
Input: $block$ - block of ordered transactions, $schedule$ - schedule with all dependencies tracked
Context:
1  procedure ExecuteBlock($block, schedule$):
2      for $i \leftarrow 0$ to $N\_THREADS$ do
3          loop
4              $tx\_id \leftarrow$ GetNextTransaction($schedule$) // Algorithm 4.8
5              if $tx\_id \neq \perp$ then
6                  $tx \leftarrow block[tx\_id]$
7                  ExecuteTransaction($tx, schedule$)
8                  OnTransactionFinish($tx, schedule$)
9              else
10                 break
11 procedure OnTransactionFinish($tx, schedule$):
12     $ready\_txs \leftarrow \emptyset$
13     $partial\_ready\_txs \leftarrow \emptyset$
14     $txs\_excluded\_from\_partial \leftarrow \emptyset$
15     $dep\_txs \leftarrow schedule.DependentTxs[tx]$
16     if $dep\_txs \neq \perp$ then
17         foreach $t \in dep\_txs$ do
18             $no\_dependencies\_left \leftarrow schedule.Deps[t]$.RemoveDependency($tx$)
19             if $no\_dependencies\_left$ then
20                 $ready\_txs \leftarrow ready\_txs \cup t$
21                 $txs\_excluded\_from\_partial \leftarrow txs\_excluded\_from\_partial \cup t$
22     $partial\_ready\_txs \leftarrow schedule.PartialReadyTxs[tx]$
23     if $partial\_ready\_txs \neq \perp$ then
24         foreach $t \in partial\_ready\_txs$ do
25             if $t \notin txs\_excluded\_from\_partial$ then
26                 $partial\_ready\_txs \leftarrow partial\_ready\_txs \cup t$
27     PushAndSignal($ready\_txs, partial\_ready\_txs, tx$) // Algorithm 4.8
```

will only focus on the functionality implemented upon these two events. After executing the transaction the thread needs to update the queues with any new transactions that might be free of conflicts (line 8). The process is straightforward - Upon finishing a transaction, threads will run through all transactions depending on the finished one (lines 16-17), and will decrease the dependency counter for each of those transactions (line 18). If the counter of a transaction reaches 0, then add it to the `ready_txs`, and add to the local set `txs_excluded_from_partial` (lines 19-21). Next, we run over all transactions whose first operation depends on the finished transaction (lines 22-24), and if this transaction was not yet added to `ready_txs`, then add it to `partial_ready_txs`. Finally we call `PushAndSignal`, which will push `ready_txs` into `QueueReady` and `partial_ready_txs` into `QueuePartialReady` in a controller manner, and will notify threads waiting on a new transaction.

We will now focus on the transaction popping and pushing mechanisms (lines 4 and 27), which are represented in Algorithm 4.8. The idea is simple - threads try to pop first from `QueueReady`. For that, threads acquire a lock for the queue, and try to pop a transaction ID from it. If any, then the status of that transaction is set to `Executing`, and this ID is returned, unlocking the queue (lines 3-6). If however the queue is empty, threads try popping from the `QueuePartialReady`, set the popped transaction's status if any, and return, unlocking the queue (lines 8-11). In case no transaction is available from any of the queues, then transactions will have to check for the termination condition (line 14) and wait if there are still transactions left to be executed. Note that a transaction may pass through the condition in line 14, if the `schedule.executed_transactions` counter is still less than `schedule.total_txs`, but the remaining transactions are already being executed by some other transaction. In such cases threads will still wait, and be notified by the final thread. This is why the lock from lines 13-16 and 33-37 is needed. We need to ensure only one thread at a time is either checking the termination condition and waiting upon it or is altering the termination condition. Else, it could happen that a thread A could pass the condition at line 14, and before falling into the wait state, thread B, responsible for executing the last transaction would execute lines 35-40, notifying all waiting threads. Since thread A was not waiting yet, it would wait only after the notification, missing the notification, waiting forever. The lock solves this issue. While popping happens before starting executing a new transaction (line 4, Algorithm 4.7), pushing on the other hand happens at the very end of execution.

As seen earlier, upon finishing execution, threads enter the `OnTransactionFinish` function in Algorithm 4.7, and at the very end (line 27), they invoke `PushAndSignal` represented in Algorithm 4.8 lines 18-45. The goal is to insert the passed `ready_txs` and `partial_ready_txs` in the respective queues and signal any waiting transactions. Ready transactions are pushed to the `QueueReady` by acquiring a lock on the queue (lines 20-25) and partial ready transactions are pushed to `QueuePartialReady` also by acquiring a lock in lines 27-31. There are however a few nuances to keep into account. For both cases, we only push if the transaction has not started executing yet (lines 22 and 29). In the case of pushing ready transactions we also need to remove the transaction from `QueuePartialReady` (line 25).

After having pushed the transactions threads acquire the `schedule.executed_txs` lock. As explained earlier, this is essential to ensure termination of the algorithm. The counter under that lock is incremented by one (line 35), and finally threads go over the termination condition. If all transactions have been executed, then notify all waiting txs so that they re-evaluate the termination condition and exit execution (lines 39-40). Else, if only two transactions were added, then the current thread will pick up one of those, and also wakes one more thread for the other transaction. If more than two transactions are added, then wake all waiting threads.

Notice the holding of the locks from popping from a queue to updating the transactions status (lines 3-5 and 8-10). This is needed to ensure atomicity of transaction status change and popping. Otherwise, a thread wanting to insert a new transaction would see that the transaction did not yet have its status as `Executing` yet, meaning it would assume that transaction was not yet inserted in the queue, and would insert it again, even though thread 2 had just popped it from the queue. With a lock, either the transaction is still in the queue, or has its state set to something other than `NotExecuted`.

Algorithm 4.8: Schedule Execution - Queue Management

```
Context: $schedule$
procedure GetNextTransaction():
    loop 'try_popping
        $ready\_tx \leftarrow$ PopQueueReady ($schedule$) // Holds a lock for this queue
        if $ready\_tx \neq \perp$ then
            SetTxStatus($schedule, ready\_tx, TxState.EXECUTING$)
            return $ready\_tx$                    // Releases the lock on return
        Unlock ($ready\_tx$)
        $partial\_ready\_tx \leftarrow$ PopQueuePartialReady ($schedule$) // Holds a lock for this queue
        if $partial\_ready\_tx \neq \perp$ then
            SetTxStatus($schedule, partial\_ready\_tx, TxState.EXECUTING$)
            return $partial\_ready\_tx$                    // Releases the lock on return
        Unlock ($partial\_ready\_tx$)
        $executed\_txs \leftarrow schedule.executed\_txs$.Lock()
        while $executed\_txs < schedule.total\_txs$ do
            Wait($executed\_txs$) // Releases the lock & waits upon signal
            continue 'try_popping
        break 'try_popping

Context: $schedule$
procedure PushAndSignal($ready\_txs, partial\_ready\_txs, tx\_finished$):
    $total\_added\_txs \leftarrow 0$
    $ready\_lock \leftarrow schedule$.QueueReadyLock()
    foreach $tx \in ready\_txs$ do
        if $schedule.tx\_stats[tx] = TxState.NOT\_EXECUTED$ then
            $total\_added\_txs \leftarrow total\_added\_txs + 1$
            PushQueueReady($tx$)
            RemoveQueuePartialReady($tx$)
    Unlock ($ready\_lock$)
    $partial\_ready\_lock \leftarrow schedule$.QueuePartialReadyLock()
    foreach $tx$ in $partial\_ready\_txs$ do
        if $schedule.tx\_stats[tx] = TxState.NOT\_EXECUTED$ then
            $total\_added\_txs \leftarrow total\_added\_txs + 1$
            PushQueuePartialReady($tx$)
    Unlock ($partial\_ready\_lock$)
    $executed\_txs \leftarrow schedule.executed\_txs$.Lock ()
    SetTxStatus($schedule, tx\_finished$, TxState.EXECUTED)
    $executed\_txs \leftarrow executed\_txs + 1$
    $has\_messages\_to\_execute \leftarrow executed\_txs < schedule.total\_txs$
    Unlock($executed\_txs$)
    // Termination condition
    if $\neg has\_messages\_to\_execute$ then
        NotifyAll($schedule.waiting\_txs$)
    else if $total\_added\_txs = 2$ then
        NotifyOne($schedule.waiting\_txs$)
    else if $total\_added\_txs > 2$ then
        NotifyAll($schedule.waiting\_txs$)
    // If only 1 or none txs are added, then no need to notify - this thread will handle it
```

Supporting Commutative Operations While we have previously examined concurrency control mechanisms at the queue level, we now shift our focus to how operation-level concurrency control manages conflicts. Before delving into the specifics of the algorithm, it is crucial to refine how exactly commutativity is handled by the system.

We will use the same example from Section 4.3. Consider an incremental commutative operation such as $A = A + 1$. This operation can be expressed as $W(A) : R(A) + 1$ in terms of reads and writes, where a read is performed on item A, followed by a write. In this case, the read is marked as commutative because it is used for the purpose of an increment, and does not require fetching the most recent value of A but can instead serve solely for delta computation. Specifically, the value obtained by $R(A)$ and the final value written can be used to compute the difference between the initial and final states, thereby deriving the delta. In this example, as the operation is additive, the delta is computed by subtracting the initial read value from the final written value. This delta is stored in the commutative write node. Therefore, commutative reads, as discussed in Section 4.3, do not depend on the most up-to-date state and can retrieve values directly from storage. However, the value read must be stored within the corresponding operation node to allow subsequent write operations to compute the delta.

Interface for Commutative Operations We have mainly been focusing on incremental operations so far, such as addition, but COEX-P is not limited to this operation type only. It supports any type of commutative operation, as long as: (1) the SE engine is capable of identifying the commutativity in all reads and writes associated with the operation from the smart contract's bytecode, and (2) the system has access to the operation's implementation of the following interface:

- `Merge`(V_{initial}, Δ): Merges a delta, Δ, into the initial value V_{initial}.
- `ComputeDelta`(V_{old}, V_{new}): Computes the delta, Δ, given an initial value V_{old} and a new value V_{new}.
- `FromBytes`(B): Converts a byte representation, B, into its in-memory value representation.
- `ToBytes`(V): Converts an in-memory value, V, into its byte representation.
- `InitAccumulatedDeltas`(): Initializes an empty container which is to be used for merging the deltas.

Having discussed operation commutativity and the interface exposed to the users of the system, we now proceed to examine the details of operation-level control (Algorithm 4.9). We begin by analyzing the write operations, as their behavior is simpler to explain. Recall that both `db_read` and `db_write` events occur during transaction execution (Algorithm 4.7, line 7). Upon receiving a write request from the WVM, the thread executing the current transaction consults the RWS context retrieved earlier (line 2) to determine whether the current write operation is commutative. If the operation is commutative, the thread fetches the corresponding commutative read from the same transaction (line 4) and employs the

Algorithm 4.9: Schedule Execution - Operation-Level Control

```
Input: $key$ - key of storage for current smart contract executing, $value$ - value to write
Context: $rws$ - ordered RWS for current transaction - also maps each operation to its corresponding schedule node
upon db_write($key, value$):
    $operation \leftarrow$ GetNextOperation($rws$)
    if $operation$ is $Commutative$ then
        $prev\_comm\_read \leftarrow schedule$.GetPrevCommRead()
        $\Delta \leftarrow$ CommOp.ComputeDelta($prev\_comm\_read.value, value$)
        $operation.value$.SetAndNotify($\Delta$)
    else
        $operation.value$.SetAndNotify ($value$)

Input: $key$ - key of storage for current smart contract executing
Context: $rws$ - ordered read-write set for current transaction - also maps each operation to its corresponding schedule
        node, $storage$ - storage for current SC executing
upon db_read($key$):
    $operation \leftarrow$ GetNextOperation($rws$)
    if $operation$ is $Commutative$ then
        $initial\_val \leftarrow storage$.Get($key$)
        $operation.value$.SetAndNotify($initial\_val$)
    else
        $starting\_node \leftarrow \perp$
        $starting\_val \leftarrow \perp$
        if $operation$ has dependency $dep$ then
            $starting\_node \leftarrow dep$
            $starting\_val \leftarrow dep.value$.WaitForValue()
        else
            $starting\_node \leftarrow schedule$.GetLinkedList($operation.sc\_addr, key$)
            $starting\_val \leftarrow storage$.Get($key$)
        $accumulated\_val \leftarrow$ CommOp.InitAccumulatedDeltas($starting\_val$)
        $node \leftarrow starting\_node$
        loop
            if $node.Tx = starting\_node.Tx$ then
                return $accumulated\_val$
            // These writes can only be commutative
            if $node$ is $Write$ then
                $val \leftarrow node.val$.WaitForValue()
                $accumulated\_val \leftarrow$ CommOp.Merge($accumulated\_val, val$)
```

interface methods to compute the delta, given both the previous commutative read value, denoted as `prev_comm_read_value`, and the new value, denoted as `value`. This computed delta is then stored in the node (line 6). Conversely, if the write operation is non-commutative, the thread directly writes `value` into the node and notifies any waiting threads dependent on this value.

For read operations, a similar procedure is followed, where the operation context is retrieved from the RWS (line 10). The behavior of the read depends on its commutativity. If the read is commutative, it does not conflict with any other operations, and the thread simply retrieves the initial value from storage (line 12), writing it into the operation node. This is the only case where a read node defines the value held by the node. However, in the case of a non-commutative read, it requires the most recent value, which may have been modified by preceding operations. This is where **explicit dependencies** come into play. The approach is to start from the most recent non-commutative write (the node to which the `dependency`

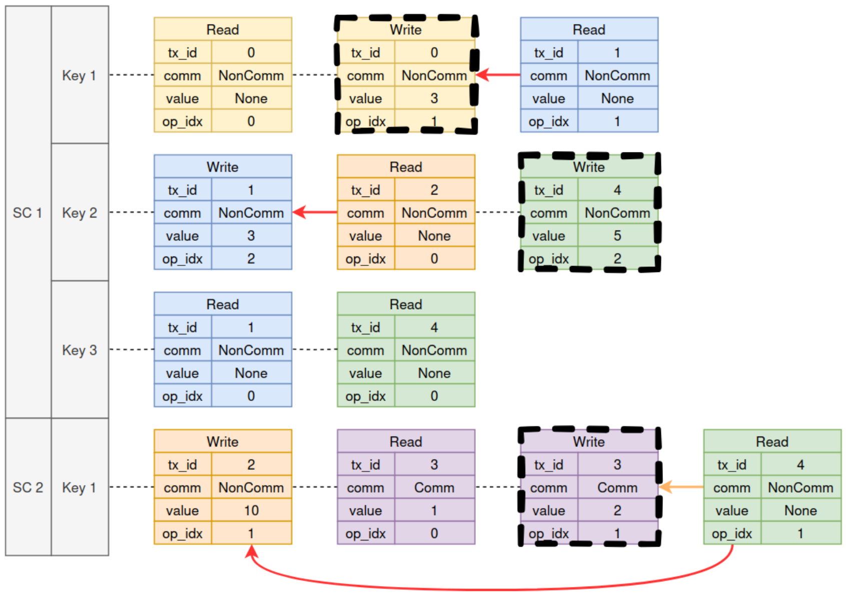

SC 1
SC 2
Key 1
Key 2
Key 3
Key 1
Read tx_id 0 comm NonComm value None op_idx 0
Write tx_id 0 comm NonComm value 3 op_idx 1
Read tx_id 1 comm NonComm value None op_idx 1
Write tx_id 1 comm NonComm value 3 op_idx 2
Read tx_id 2 comm NonComm value None op_idx 0
Write tx_id 4 comm NonComm value 5 op_idx 2
Read tx_id 1 comm NonComm value None op_idx 0
Read tx_id 4 comm NonComm value None op_idx 0
Write tx_id 2 comm NonComm value 10 op_idx 1
Read tx_id 3 comm Comm value 1 op_idx 0
Write tx_id 3 comm Comm value 2 op_idx 1
Read tx_id 4 comm NonComm value None op_idx 1

Figure 4.12: Schedule Persistence - Black dashed borders identify the last writes for each key.

field points to), if such an operation exists, or from the beginning of the operation sequence (lines 17-22). Specifically, the algorithm begins by checking the **explicit dependency** of the node to obtain the initial value. If no such dependency exists, it starts from the head of the operation list, using the value from storage as the base.

Subsequently, the algorithm iterates over all operations, waiting for the values to be written by preceding write nodes, if necessary, using condition variables. As the iteration progresses, the thread merges any commutative deltas encountered (lines 29-31). By design, since the iteration begins at the last non-commutative write with a transaction ID lower than that of the current transaction, or from the head of the list in the absence of such a write, all subsequent writes must be commutative. Therefore, the algorithm iterates through and merges all potential deltas into the initial value. This allows the transactions responsible for those commutative operations to execute those operations in parallel, while ensuring that a non commutative read comming afterwards will always wait for those deltas before returning the value. The final result, after all merges have been completed, is then returned to the caller, namely the WVM.

Upon finishing execution, the system enters its final phase - Schedule Persistence.

4.7 Persisting the Schedule

Running Example After all threads are finished executing, the final step is to persist the last written values from the schedule into the SC's storage. This step is simple enough to be run single-threaded. The idea is depicted in Figure 4.12. Black dashed borders represent the last write nodes, which will be

Algorithm 4.10: Schedule Persistence

```
Input: $key$ - key of storage for current smart contract executing, $value$ - value to write
Context: $storage\_manager$ - manages storage for each contract
procedure PersistSchedule($schedule$):
    foreach $contract$ in $schedule$ do
        $sc\_storage \leftarrow storage\_manager$.GetContractStorage($contract$)
        foreach $key$ in $contract$ do
            $last\_write \leftarrow schedule$.GetLastWrite()
            if $last\_write \neq \perp$ then
                if $last\_write$ is $Commutative$ then
                    $val \leftarrow$ MergeLastCommutativeWriteChain($last\_write, sc\_storage, key$)
                    $sc\_storage$.SetValue($val$)
                else
                    $val \leftarrow last\_write.value$.WaitForVAlue()
                    $sc\_storage$.SetValue($val$)

Input: $key$ - key of storage for current smart contract executing, $value$ - value to write
Context: $storage\_manager$ - manages storage for each contract
procedure MergeLastCommutativeWriteChain($last\_write, sc\_storage, key$):
    $accumulated\_deltas \leftarrow CommOp$.InitAccumulatedDeltas()
    $node \leftarrow last\_write.prev$
    loop
        if $node$ is $Write$ then
            $val \leftarrow node.value$.WaitForValue()
            $accumulated\_deltas \leftarrow CommOp$.Merge($accumulated\_deltas, val$)
            if $node$ is NonCommutative then
                return $accumulated\_deltas$
        if $node.prev \neq \perp$ then
            $node \leftarrow node.prev$
        // Reached head of operations without having found a non commutative write - use storage value
        else
            $val \leftarrow sc\_storage$.Get($key$)
            $accumulated\_deltas \leftarrow CommOp$.Merge($accumulated\_deltas, val$)
            return $accumulated\_deltas$
```

iterated over and used to compute the value that is to be persisted into the corresponding SC storage. Following the example, and going from top to bottom, the first storage key to be persisted would be `SC1`, `Key1`. The last write for this key is the one from `Tx0`, which has written the value 3 to its node. Since this write is non-commutative, this is the final value persisted to the `SC1`'s storage for `Key1`. Next, for `Key2` we have another non-commutative write as the last write. The process is similar - we persist the value written by that operation. In this case it would be 5. Next, for `Key3`, since it has not writes, no modification to the initial storage is needed.

Finally, for `SC2, Key1` the last write is a commutative operation. Since this operation only tracks the delta, and not the last value, we need to iterate over all nodes to the left until we find the first non-commutative write, or we reach the head of the list of operations. In this example we would reach the orange write, fetch its value - 10, and merge the delta 2, which would produce a final value of $10 + 2 = 12$. This would be the value to be persisted for `SC2`,`Key1`.

Schedule Persistence Upon finishing execution, threads follow the `PersistSchedule` procedure, as shown in Algorithm 4.10. They iterate over each SC in the schedule, fetching its storage (line 3). Then they iterate over all keys of that SC and fetch the last write (line 5). If there is no last write, then that key's value was not changed, i.e., no work needs to be done. If there is a last write and it is commutative (line 7), the thread needs to go over all previous deltas and merge them as represented in by the procedure `MergeLastCommutativeWriteChain` in line 13. If, on the other hand, the last write for the current key is non-commutative, the thread only has to persist the value written in that node (lines 10-12).

For commutative writes (line 8) the thread initializes, using the commutative operation's interface defined earlier, a base value that will be used to merge the deltas (line 14). Then it loops from right to left following the linked list of operations. Upon finding a write, it waits for the delta of that node and then merges it (lines 17-19). Note that this value being merged is only a delta if the write is commutative. If the write is not commutative, then it has reached the end of the commutative chain. It still merges the deltas to the value of the first non-commutative write, and then returns the merged value (lines 20-21). If no non-commutative write is ever found, then the thread reaches the head of the linked-list. In such case it reads the value from storage, merges the deltas to that value and returns the merged value (lines 25-28).

After persisting the storage, the system ends its work, and is ready to receive a new block, construct a new schedule for it, and execute the entire pipeline again.

4.8 Conclusion

In this Chapter we introduced COEX-P, a novel approach to optimizing the execution of smart contracts through parallel execution of commutative operations. By leveraging scheduled execution with commutative operation support, we aim to address several key challenges in smart contract execution. Key contributions of our approach include:

1. Efficient conflict resolution: By building with write-versioning in mind, we allow conflicting operations to be processed concurrently in a deterministic way.
2. Scalability: The use of a parallel schedule building together with a parallel schedule execution allows the system to handle a large number of transactions efficiently.
3. Hot-spot awareness: By relying on operation commutativity support, the system is designed to perform efficiently under commutative-intensive workloads.

5 Evaluation

This section provides a comprehensive evaluation of the proposed solution, focusing on three key aspects: *correctness*, *performance*, and *resource utilization*. The system is benchmarked against a baseline, namely, the standard sequential execution, to demonstrate the benefits of parallelization. Additionally, the base parallel version without commutativity support is compared against the system with commutative operation support to assess the impact of operation commutativity. The evaluations are designed to stress-test the system across various scenarios and workload patterns, including the challenging hot-spot scenario.

Section 5.1 evaluates the system's performance across different scenarios. Section 5.2 examines the system's resource utilization, particularly hardware resources, under varying workloads. Finally, Section 5.3 concludes with an analysis of the key improvements introduced by the system, as well as a discussion of the current limitations evaluation results show.

Correctness Evaluation To validate correctness, the system was tested by computing a hash representing the final state across all smart contract (SC) storages in the vanilla sequential execution, and comparing it to the final state produced by the proposed solution.

In all following experiments, the resulting state was the same, confirming that the output of the proposed system is equivalent to that of the serial execution model.

As we will discuss later, the experiments were made using a **serial** schedule creation. Initial implementations of the system benefited from the parallel version introduced in Section 4.5, but as of now, the current version seems to produce lower and more consistent times when building the schedule serially.

5.1 Performance Evaluation

This section addresses the following key performance questions:

1. How does the system's performance scale with varying workloads and thread counts?

2. What are the overheads associated with schedule creation and persistence, and how do they vary with different workloads?
3. How does the support for operation commutativity compare to the base parallel version?

The evaluation begins by analyzing the system's performance as the number of threads increases, comparing it to the sequential baseline, and measuring the time spent on schedule creation and persistence. The following tests explore the impact of varying several workload characteristics, including:

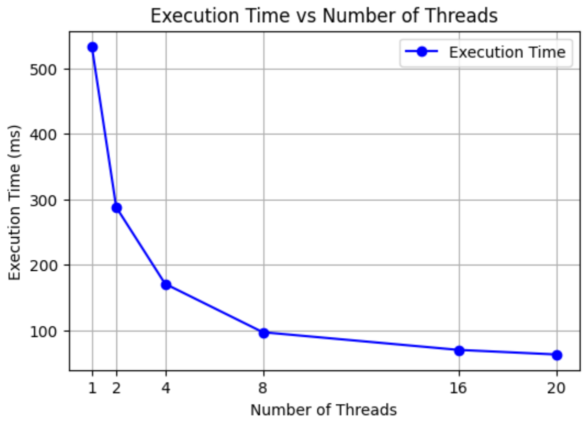
1. Transaction complexity
2. Number of smart contracts (SCs)
3. Number of transactions per SC
4. Block size
5. Presence of commutative operations

Transaction Logic The system processes smart contract transactions, each performing a sequence of read and write operations to storage. For the purpose of this evaluation, a synthetic SC was built to simulate common operations - an increment on a global counter, and a generic storage set operation that conflicts with any other read or write to that same storage key. Using synthetic smart contracts allows for a more controlled and repeatable testing environment. By simulating specific operations, the results show more uniform behavior across evaluations, making it easier to isolate and analyze system performance under various conditions. Transactions vary in terms of:

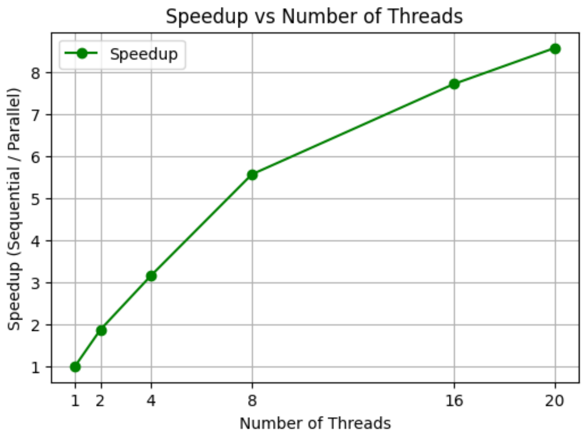
- **Complexity:** Amount of work done between the reads and the writes. Each of the available SC's functions loops over N iterations, simulating transaction computation. This is a variable used to stress the system on simple vs. complex transactions.
- **Commutativity:** The synthetic SC constructed for the tests allows calling two functions - one that increments a global counter, and one that just sets the value of that counter. This means transactions can be commutative in the case of calling the incremental method or non-commutative in the case of just setting the storage with some predefined value.

The experiments will also vary the amount of conflicts present, by varying the amount of transactions to the same smart contract.

To ensure robust results, each test was executed five times, and the mean execution time was computed for each configuration. The experiment excludes any time spent on instantiating the contracts, and only captures transactions invoking functionality on already deployed SCs. The experiments were



(a) Execution time



(b) Speedup

Figure 5.1: Thread Count - Execution time and speedup.

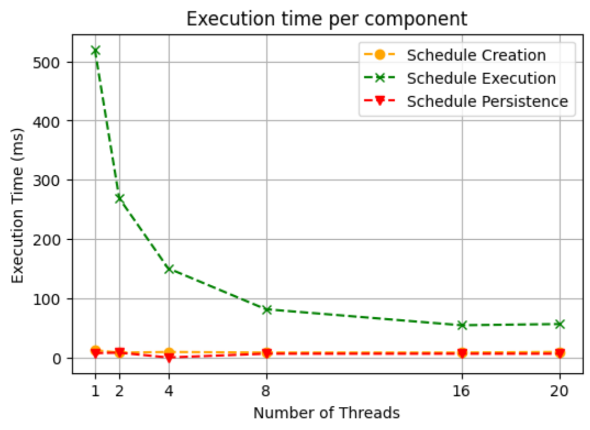
conducted on a Linux server equipped with an Intel(R) Xeon(R) Gold 6138 @ 2.00Ghz (single socket with 20 cores) and 64GB of RAM. The number of threads used in the experiments was varied from 1 up to 20 to fully utilize the available hardware resources.

5.1.1 Performance Scalability

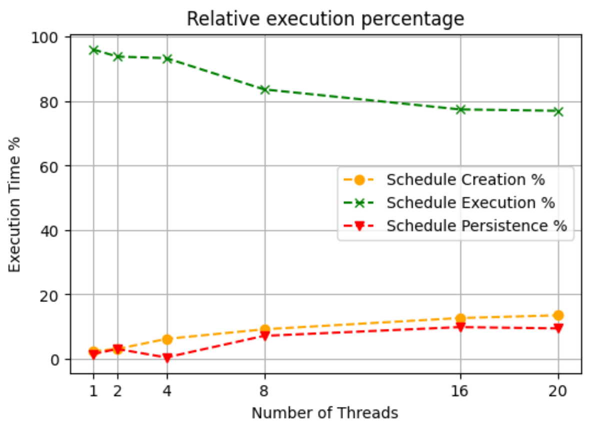
Thread Count To understand how performance scales with an increasing number of threads, the next experiment utilized a workload comprising 200 SCs, each containing 10 transactions and a block size of 2000 transactions. Each transaction reads a value from storage, simulates some computational logic, and then writes the final result non-commutatively to the same storage key from which it was read. Although transactions within each SC may conflict with one another, since the number of SCs is much larger than the number of threads, there will always be ready transactions to be executed.

The complexity of each transaction, represented by the number of loop iterations that apply a sum to a variable, was set to $300,000$. This implies that, between the read and subsequent write operations, each transaction performs $300,000$ iterations. We measured the execution time for a single transaction with $300,000$ loop iterations and it took well below one millisecond. This level of complexity was intentionally chosen, as it reflects the most common scenarios in real-world workloads where computational efficiency is paramount. Note that, as mentioned, later we will present a sensitivity analisys in which the number of loop iterations varies across a relatively large range.

Figure 5.1 illustrates the execution times on the left, measured in milliseconds, and the corresponding speedups for each thread count, relative to the sequential execution baseline. The results clearly indicate a significant reduction in block execution times with each increment in the number of threads, decreasing from approximately $540\,$ms to only $60\,$ms. Furthermore, the observed speedups continue to increase with



(a) Execution time



(b) Relative execution time percentage

Figure 5.2: Thread Count - Absolute and relative per-component execution times.

the addition of threads, albeit not in a linear manner. The maximum speedup achieved for this workload was $8.57\times$, which is slightly below half of the optimal speedup theoretically attainable with 20 threads.

The following experiments will help explain that this phenomenon can be attributed to various factors, including the selection of the VM for transaction execution and concurrency overhead at the ready queue level. Given that transactions have very brief execution times, threads frequently access the queues, resulting in an higher overhead related to concurrency control.

Figure 5.2 presents a per-component analysis of the system's performance for the same workload. As explained in Chapter 4, schedule creation refers to creation of the schedule. Schedule execution refers to the execution of transactions including all concurrency control. Schedule persistence refers to iterating over all items and persisting them into each SC's storage. Figure 5.2 (a) illustrates the absolute execution times for each phase of the execution pipeline. It is evident that only the schedule execution phase is decreasing in execution time, while both schedule persistence and creation remain constant. This observation aligns with the expectations, given that both components operate sequentially and the workload parameters—including the number of SCs and total operations—remain unchanged. Conversely, Figure 5.2 (b) showcases the relative execution percentages attributed to each component. Despite schedule creation and persistence being constant, their relative execution percentages increase as the number of threads rises, since the schedule execution phase continues to consume a smaller fraction of the overall execution time, ultimately accounting for less than 80%. In contrast, both schedule creation and persistence occupy approximately 15% to 17% of the total execution time.

Block Size The next set of experiments aims to investigate how the system performs with different block sizes. We tested block sizes ranging from 200 to $10,000$ transactions, while maintaining the same conditions as in the previous experiment: 200 SCs and $300,000$ loop iterations, focusing solely on non-

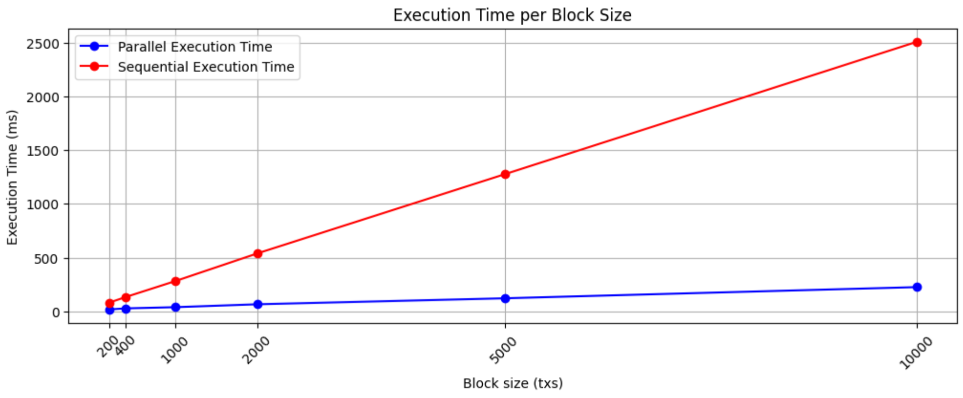


Figure 5.3: Block Size - Execution times

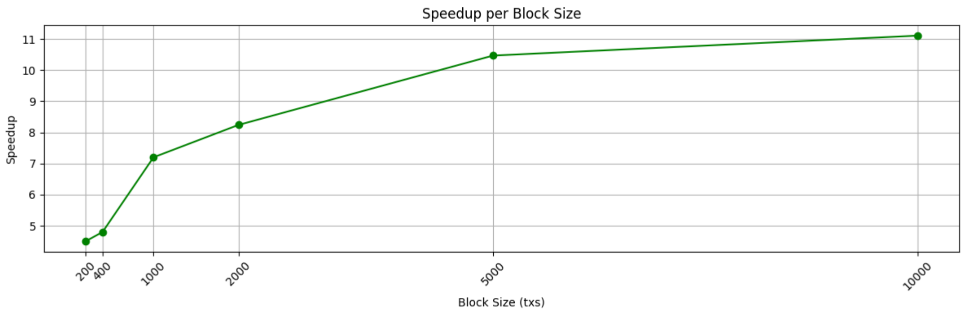


Figure 5.4: Block Size - Speedup

commutative operations. This time, we fixed the number of threads to 20, as this setting yielded the best performance in our prior tests.

Figure 5.3 illustrates the mean execution time per block as we vary the block size. Notably, while the sequential version's execution time doubles each time the block size is doubled, the parallel version experiences a much slower rate of increase. For instance, with a block size of $10,000$ transactions, the sequential execution time was around 2.5 s, whereas the parallel execution took just over 200 ms. As shown in Figure 5.4, the speedup tends to increase with block size. This trend occurs because, as the following plots reveal, the relative execution times for both schedule creation and persistence decrease, allowing the parallel components of the system to become more significant. As a result, the overall speedup is primarily influenced by the schedule execution, which efficiently parallelizes and drives the system's performance.

Figure 5.5 presents the execution time of each system component as the block size increases. Two notable observations emerge from the data: first, both the execution phase and schedule creation times tend to increase, which is expected since more transactions lead to more operations that need to be scheduled and executed. Second, the schedule persistence time remains constant. This is anticipated, as the experiment does not alter the number of keys being written but rather changes the number of

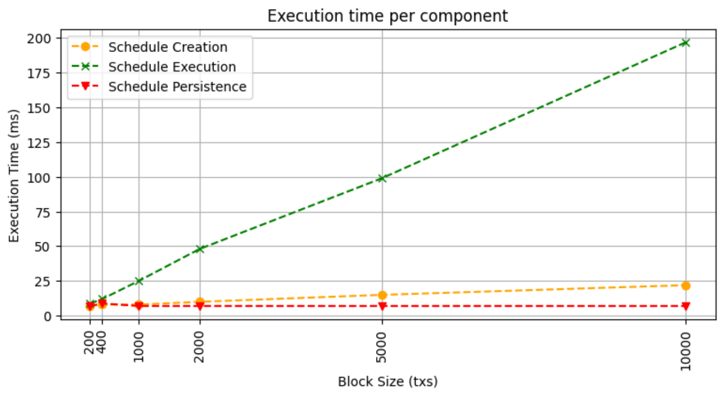


Figure 5.5: Block Size - Execution times

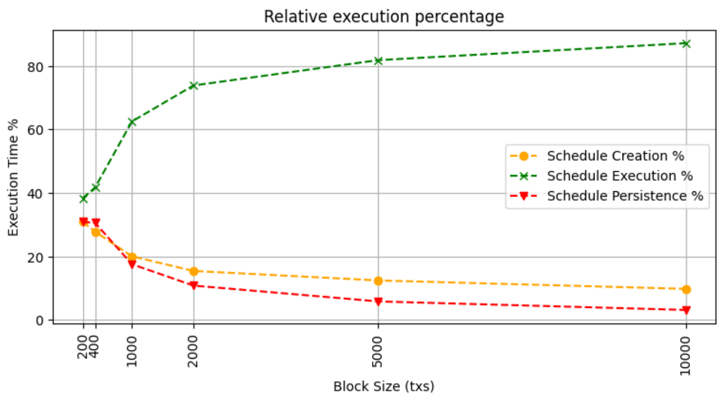
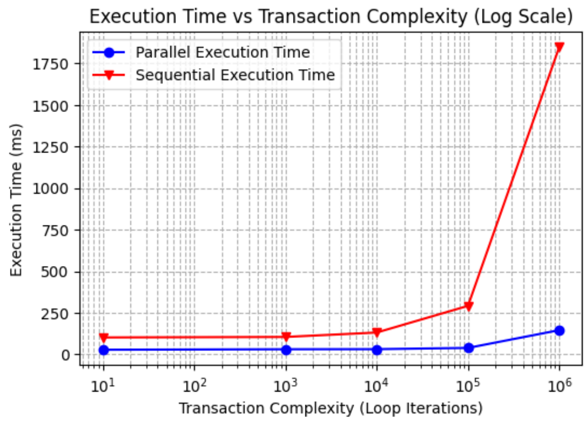


Figure 5.6: Block Size - Relative execution time percentage

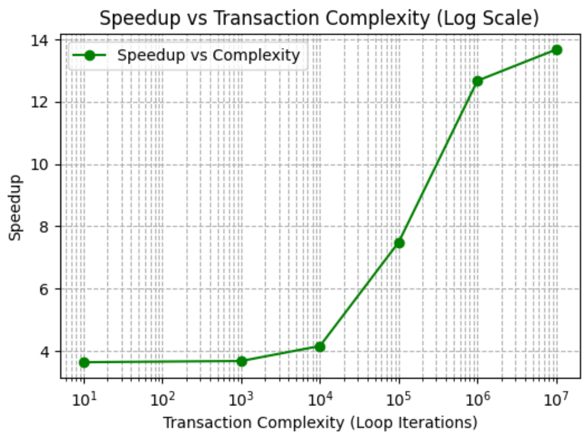
transactions writing to them. Thus, the number of keys that the schedule persistence phase must iterate over stays the same. Interestingly, for smaller block sizes (ranging from 200 to $1,000$), the combined times for schedule creation and persistence account for approximately 60% of the total execution time, while the execution phase comprises only 40%. As depicted in Figure 5.6, this indicates that without further optimizations to the schedule creation and persistence processes, the system's speedup is hindered with smaller block sizes, achieving only about $4.8\times$ the sequential execution speed with 20 threads. If we could reduce the time taken by schedule creation and persistence to half of their current durations, the speedup could potentially increase to around $7\times$. This improvement could stem from simply reducing the constant time of 8 ms each currently takes, which for smaller blocks could significantly enhance the speedup, possibly by over $2\times$.

5.1.2 Performance over Different Workloads

Transaction Complexity Transaction complexity controls the time each transaction spends executing. Consequently, it is expected that higher complexity leads to increased execution times for transactions



(a) Execution times



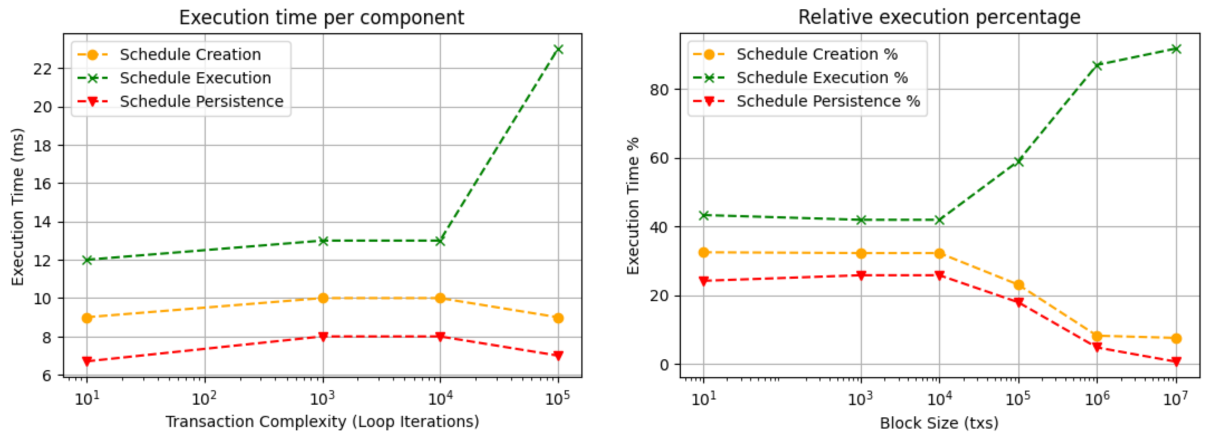
(b) Speedup

Figure 5.7: Transaction Complexity - Execution time and Speedup

and decreased time spent on concurrency control. Therefore, speedups are expected to increase with an increase of complexity. This experiment aims to confirm this expectation while providing additional insights into current system bottlenecks for lower transaction complexities. The test configurations remain the same as before: 200 smart contracts (SCs) with 10 transactions each, with a block size of $2,000$ transactions.

Figure 5.7 illustrates the execution times in milliseconds and the corresponding speedups against the sequential version for different tested transaction complexities. The system was tested from only 10 loop iterations to over 10^7 iterations. (The highest complexity value used is not displayed in Figure 5.7 (a) to avoid scaling the time axis too much, which would hinder readability for lower complexities). However, since transactions in real workloads tend to be highly efficient and exhibit low instruction complexity, the block execution time generally remains sub-second. Therefore, complexities around 10^5 and below are of particular interest. The parallel version tends to take around 30 ms for complexities below 10^5, while the sequential version consistently exceeds 100 ms. Speedups range from $3.6\times$ to $7.5\times$ at 10^5 loop iterations, clearly indicating that the overhead is much more pronounced at such low complexities, and also that more complex contracts will result in better performance.

Figure 5.8 illustrate this overhead for lower complexities in a per-component basis. In Figure 5.8 (a), it is evident that for very low transaction complexities, the time spent on schedule creation and persistence is comparable to that of schedule execution (both around 10 ms). This implies that schedule execution is responsible for only about 40% of the total execution time, clearly highlighting schedule creation and persistence as bottlenecks for workloads with low complexities, as shown in Figure 5.8 (b). This pattern was also evident in the previous experiments with varying block sizes, as illustrated in Figure 5.6. However, as we will discuss next, this pattern tends to disappear when the workload transitions from *sparse* to



(a) Execution time

(b) Relative execution time percentage

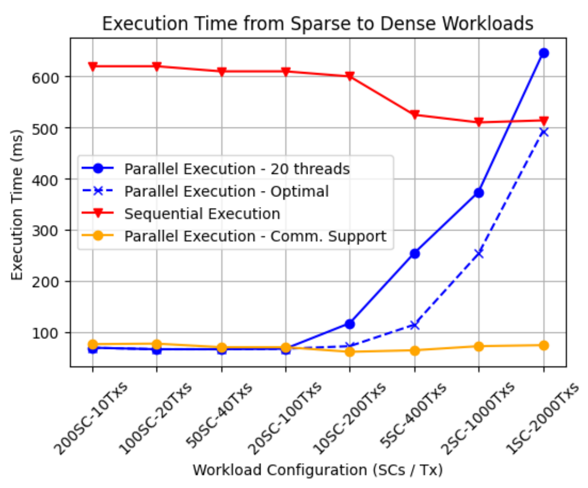
Figure 5.8: Transaction Complexity - Absolute and relative per-component execution times

dense. Specifically, this transition occurs when the workload shifts from having a significantly larger number of SCs compared to the number of transactions to having a smaller number of SCs relative to the number of transactions. This scenario is expected for hot-spot access patterns, where numerous transactions affect only a few keys.

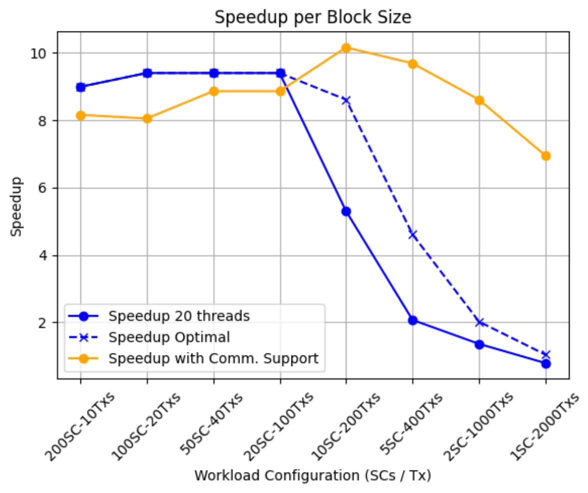
Dense Workloads with Commutativity Support Although previous experiments show how the system behaves for sparse workloads, i.e., workloads with a high number of SCs, each with a few transactions, this next experiment shows how the system performance varies when the workload changes from being sparse, to dense, as this is the common hot-spot scenario - a large number of calls to only a few SCs, touching only on a few keys. Furthermore, dense workloads benefit most from commutativity support as they are inherently conflict-intensive. The experiment assumes a block size of $2,000$ transactions, using 20 threads (the maximum available amount), and fixes the total number of transactions to $2,000$, and varies the SC-Tx relationship, going from 200 SCs, each with 10 transactions, to a single SC with $2,000$ transactions. The experiment also compares the system's operation commutativity support by running all workloads with and without commutativity support. All transactions are assumed to commute when operation commutativity is enabled, and to conflict when it is disabled.

Figure 5.9 (a) plots the execution times of four different tested cases:

1. Sequential Execution: Represented as the red line, represents the sequential execution times. As shown, the execution times decreases when the workload switches to from sparse to dense. One possibility is due to cache's principle of spacial and temporal locality. Since there are less SCs for dense workloads, threads will re-use the VMs of the same small set of SCs, whereas with sparse workloads, threads need to constantly fetch different VMs for the different SCs.



(a) Execution time



(b) Speedup

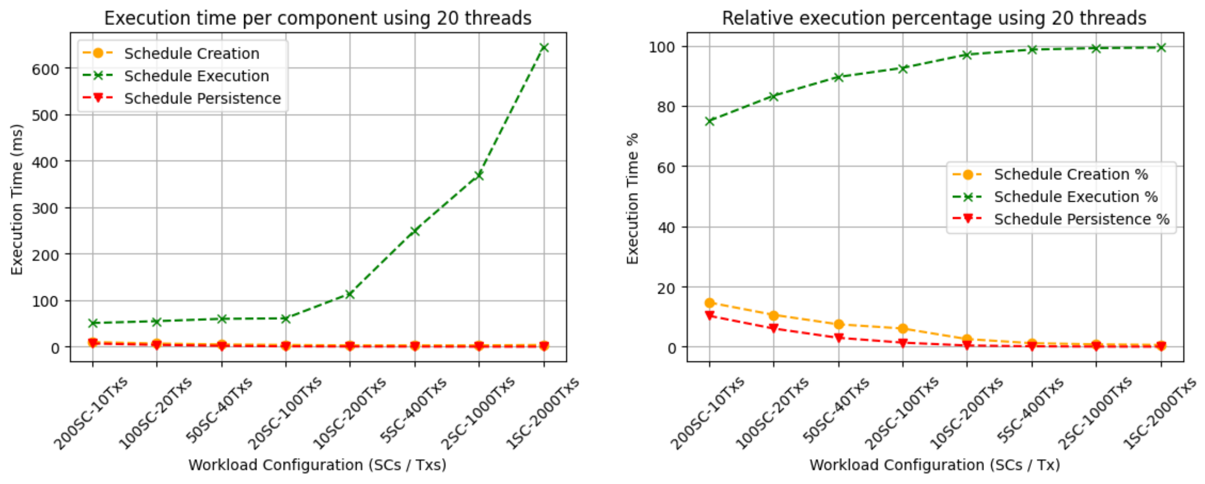
Figure 5.9: Dense Workloads with Commutativity Support - Execution time and Speedup

2. Parallel Execution with 20 threads: Represented as the blue filled line, represents the parallel execution without commutativity support, i.e., every transaction conflicts with the previous one. The plot shows that execution times tend to increase when the workloads tends to be dense. This is expected as, since transactions conflict, the number of SCs limits the degree of parallelization. For 1 SC, the maximum parallelization achievable is 1. For 2 SCs, the maximum speedup achievable is 2, and so on. This is the reason why for less than 20 SCs, when using 20 threads, execution times increase, mainly due to concurrency control. This configuration surpasses the serial version when all calls are made to a single SC.

3. Optimal Parallel Execution: Represented as the dashed blue line, represents the parallel execution, but setting the number of threads dynamically with the number of SCs. For 10 SCs, the system runs with 10 threads. For 5 SCs it runs with 5 threads and so on. This is to get rid of the high concurrency control for 20 threads, and capture the system's maximum achievable speedups. Clearly, as shown in Figure 5.9 (a) this is the reason why the execution times are always smaller than the parallel version with 20 threads.

4. Parallel Execution with 20 threads and Commutativity Support: Represented as the orange line, represents the parallel execution with operation commutativity support enabled. This is the most interesting configuration for this experiment. As shown in Figure 5.9 (a), the execution times with operation commutativity enabled are mostly constant at around 70ms with a slight increase for less than 2 SCs. This is because the experiment was set to allow all operations to commute, i.e., they can be executed in parallel, and the system only needs to merge these deltas at the end.

Looking at Figure 5.9 (b), which plots the speedups relative to the serial version using the execution



(a) Execution time (b) Relative execution time percentage

Figure 5.10: Dense Workloads with Commutativity Support - Per-component absolute and relative execution times for the parallel version **without** commutativity.

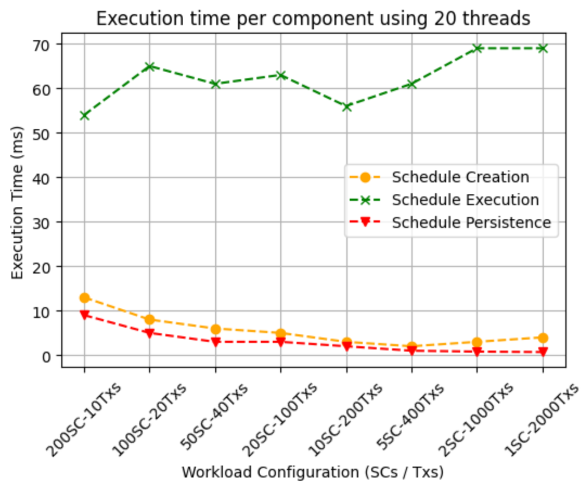
times from Figure 5.9 (a), it becomes clear that:

1. The speedup of the commutative support version falls slightly below the parallel version without commutative support for sparse SCs (for over 20SCs). One reason for this is that the current commutativity support implementation currently needs to do extra work at each node everytime the value is to be written to the node. This work includes deserializing the storage value from a byte array to the in-memory type defined by the interface in Section 4.6, computing the delta for that in-memory value, and serializing again the value into a byte array to be stored at the node. The parallel version without comutativty support avoids all this extra work, and only needs to store the byte array directly.

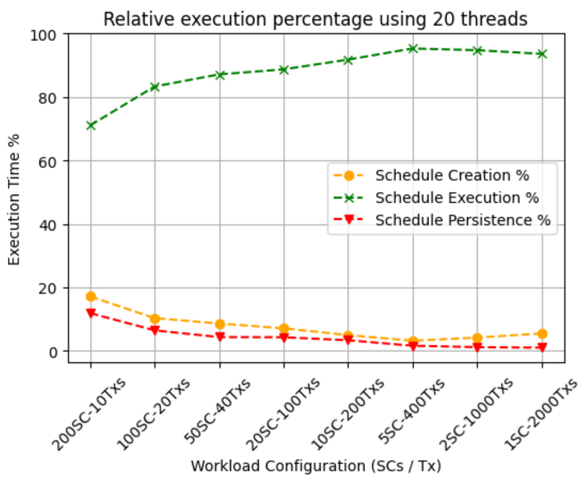
2. The speedup of the commutative support version decreases much slower than the parallel version without commutative support for dense workloads (for less than 20SCs). The reason for this is that commutative operations are allowed always to execute in parallel. The system initializes a number of VMs corresponding to the number of threads, and then, when executing, multiple threads can fetch different VMs and execute the commutative operations in parallel. This shows that, with commutativty support enabled, the system is able to perform up to $6.6\times$ better than the base parallel version for the case of multiple calls for the same SC, which confirms the benefits of commutative support under hot-spot scenarios.

Figure 5.10 and Figure 5.11 show the execution time of each component of the system as well as its relative execution percentage.

Figure 5.10 (b) and Figure 5.11 (b) plot the relative execution percentages, while Figure 5.10 (a) and



(a) Execution time



(b) Relative execution time percentage

Figure 5.11: Dense Workloads with Commutativity Support - Per-component absolute and relative execution times for the parallel version **with** commutativity.

Figure 5.10 (b) plot the execution times. Figure 5.10 show data from the parallel version with 20 threads without commutative support, and Figure 5.10 show data from the same parallel version with commutative operation support enabled. Figure 5.10 (a) shows that the main overhead, as explained earlier is due to the schedule execution, namely due to concurrency control as there are more threads than SCs the denser the workload gets. This is still the case for the version supporting commutative operations, but at a much smaller scale - while without operation commutativity the execution phase tends to keep increasing, reaching over 600ms, in the version with commutative support the execution phase was always under 70ms. This leads to a less extreme limit on the relative execution time, as shown by Figure 5.10 (b) and Figure 5.11 (b). Still, both plots show that the execution phase is still dominant, while schedule creation and persistence have a decreasing trend. This is expected as there will be less SCs, thus less items to persist. The reason for the decrease in schedule creation time may be related to how it is implemented, and seems proportional to the number of different keys inserted in the schedule.

5.2 Resource Utilization

Resource utilization was tested in terms of per-CPU and RAM usage. Since all instantiated VMs are stored in-memory, the system uses a substantial amount of memory. For 200SCs running with 20 threads the system consumed around 32GBs. Recall that each thread has an associated VM. This means that in total the system needs $200 * 20 = 4000$ VMs to fully support intra-SC concurrency for commutative operations. Initially, the system used a queue to store VMs such that threads could pop a VM at need. This allowed having less VMs than threads. Meaning the system could be launched with 20 threads and only have 5 VMs initialized for each SC. This decreases the usage of RAM by a factor of 4x, but it also impacted performance, sometimes by a factor close to 2x slower. The suspicion is that by having threads share VMs between each other, when executing transactions aimed at the same SC, threads could execute two consecutive transactions using two completely different VMs, meaning an entire VM context would need to be loaded into cache. This behavior affected performance. Furthermore, with less VMs than threads, the maximum parallelization achievable for commutative intensive workloads is limited by the number of VMs instead of the number of threads. In practice, this means that if we only have commutative operations for a single SC, and we execute with 20 threads, but only 2 VMs, the maximum speedup achievable will be $2\times$.

Instead, the current solution opts to assign each different thread a different VM, where all transactions for some SC picked by some thread are executed always under the same VM. This adheres to the principle of memory locality, improving the overall performance. This way the maximum speedup achievable is set to the number of threads instead of the number of VMs at the cost of higher memory usage.

As for CPU usage, an experiment was built to stress out the efficiency of CPU utilization. The experiment comprised of a workload with 30SCs, each with around 8000 transactions to stress out the system usage. The goals of the system are twofold:

- Utilize all available resources at maximum.
- Be efficient when it is not possible to utilize all resources due to conflicts.

Since the Linux system used so far has 20 cores, it would be hard to visualise and represent per-CPU usage. As such, this experiment was conducted using a smaller machine with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (single socket with 6 cores) and 16GB of RAM. Table 5.1 describes each tested workload.

Workload	**Description**
Workload A	30SCs each with 8000 transactions.
Workload B	4SCs each with 8000 transactions.
Workload C	2SCs each with 8000 transactions.
Workload D	1SC with 8000 transactions.

Table 5.1: Description of Tested Workloads

Each of these workloads used the system configured to use all 6 threads available. Since each transaction in this workload depends on the previous on the block, each thread, upon finishing executing a transaction should only be able to free one other transaction. As such, as explained in Section 4.6, this thread should not notify any other waiting thread, and should continue executing the chain of transactions by itself. Table 5.2 shows the CPU utilization for each of the workloads. Each entry in the table was captured using the `mpstat` tool to capture per-CPU usage every second while the system was running, after all VM initializations until the execution ended. The following table shows CPU percentage as well as the standard deviation for each CPU.

Workload	**CPU Usage (%) per vCPU**					
	1	2	3	4	5	6
Workload A	98.75 ± 0.83	98.48 ± 2.62	100.00 ± 0.05	100.00 ± 0.03	99.00 ± 1.73	99.75 ± 0.43
Workload B	99.00 ± 0.82	100.00 ± 0.05	100.00 ± 0.05	0.99 ± 0.81	2.68 ± 0.47	100.00 ± 0.05
Workload C	97.09 ± 5.56	95.21 ± 15.28	1.25 ± 0.92	1.00 ± 0.58	1.33 ± 0.94	1.00 ± 1.15
Workload D	98.00 ± 0.73	1.62 ± 1.54	1.60 ± 0.464	0.80 ± 0.57	2.91 ± 0.74	1.70 ± 0.67

Table 5.2: CPU Usage (%) per vCPU for Different Workloads

As expected, for Workload A, the system fully all 6 cores as there are 30 SCs, allowing a maximum parallelization of 30, only limited by the 6 hardware cores. For Workload B, since the number of SCs was 4, the systems uses only cores 1, 2, 3 & 6. Similarly for Workload C, since there are only 2 SCs, the system uses only 2 cores (1 & 2). Finally for Workload D the system uses a single core (1). Again, recall that since each transaction only frees one other transaction, threads that are put to sleep waiting for a new push to ready or partial ready queue will never wake up. This is an extreme case used to test out the best-case scenario of a chain of conflicting transactions, and how the system would behave. Of course, if commutative operation support was enabled, then the system would use all 6 cores even for Workload D.

5.3 Discussion

The results presented in this study described the performance characteristics of the system across various workload scenarios. They shows that the system incurs significant overhead when processing small blocks or executing transactions with low complexity. This overhead can primarily be attributed to two factors: (A) schedule creation and persistence, and (B) concurrency control mechanisms.

(A) Schedule Creation and Persistence While schedule creation and persistence typically take under 8 ms, their overhead becomes significant when transaction execution times are lower. Parallelization of these stages could offer optimizations. Early attempts to partition and merge sub-schedules improved performance, but later system optimizations made the serial version more stable. However, future iterations could still explore parallelizing these stages to reduce system overhead.

(B) Concurrency Control Overhead Concurrency control adds notable overhead, especially in low-complexity workloads due to queue-based resource management. Reducing this overhead through more efficient techniques could improve system performance. Exploring lightweight control mechanisms or minimizing synchronization points may offer further gains.

One of the key insights from this study is the significant impact of operation commutativity support in dense, conflict-prone workloads, where many transactions target a small set of smart contracts and keys. In such scenarios, commutativity support improved execution times by up to $6.6\times$ compared to the baseline parallel version without commutativity. This highlights the role of commutativity in improving performance for workloads characterized by high transaction contention. Additionally, the inherent locality of access in hot-spot workloads improves cache utilization, as frequently accessed contract's VMs remain in cache.

However, some overhead remains in the current commutativity implementation, particularly when the number of available parallel tasks equals or exceeds the number of threads. This performance degradation comes from the need to deserialize, compute deltas, and serialize state changes. A possible optimization is to store in-memory values directly in the node, rather than as serialized byte arrays, which would halve the work required during transaction execution. Additionally, deferring the serialization process to the schedule persistence phase, where in-memory values could be merged and serialized, would further reduce the overall workload and alleviate the bottlenecks associated with state management in commutative operations.

Despite these overheads, the system demonstrates a good speedup, especially considering the lightweight nature of typical blockchain workloads, which generally feature sub-second execution times. Achieving high levels of speedup under these conditions is challenging, as the relative proportion of sequential operations increases. Nonetheless, the system achieved a maximum speedup of $8.57\times$ for a block size of $2,000$ transactions with low transaction complexity under a sparse workload, and a maximum speedup of $6.9-10.1\times$ for a hot-spot workload touching 1 to 5 SCs with commutativity enabled, indicating that the parallel execution model is effective within the tested parameters.

The system has also proven efficient in its use of CPU resources. The dynamic thread management ensures that threads remain idle when no parallelizable transactions are available, thus avoiding unnecessary concurrency control overhead. This adaptive thread utilization strategy ensures that available computational resources are maximized, without introducing excessive overhead from idle threads waiting on work.

6 Conclusion

Blockchain technology has established itself as a fundamental pillar for decentralized applications across diverse domains, merging the concepts of distributed systems and databases into a cohesive framework for secure transaction management. The appearence of smart contracts has significantly expanded the capabilities of blockchain systems, enabling automated and conditional logic to execute in a trustless environment. However, the increasing complexity and popularity of these applications have exposed inherent limitations in transaction execution, particularly in high-contention scenarios characterized by hot spots.

This Thesis addresses the critical challenge of parallelizing transaction execution with a focus on the hot-spot access pattern - a large number of transactions to a small number of popular smart contracts. By focusing on the commutativity of operations, we propose a novel strategy that effectively manages hot-spot scenarios. Our approach capitalizes on the unique properties of commutative operations to minimize contention and optimize performance in environments where certain items experience high transaction volumes.

The system is configurable in terms of number of threads, operation commutativity support, and transaction reordering to achieve the best performance possible. As the experiments have shown, the system is able to achieve up to $8.57\times$ less time to execute than the sequential version, when executing without operation commutativity, and up to $10.1\times$ with operation commutativity enabled, while being up to $6.6\times$ faster then the parallel version without commutativity for dense workloads.

6.1 System Limitations and Future Work

Some of the **system's limitations** were already discussed in Section 5, when discussing the experimental results. We highlight them once more:

- Schedule creation and persistence incur significant overhead for low complexity workloads. This occurs for both small block sizes as well as transactions with low complexity.

- Concurrency Control at queue level. Results show that for increasing transaction complexity speedups increase. As schedule creation and persistence remain constant, and schedule execution takes over 90% of the relative execution time, this means the lack of further speedup increases is mostly due to the execution phase. Further timing tests also showed that the average time spent waiting for other operations to complete is also negligible, meaning the overhead must be tied to concurrency control at the queue level.
- High RAM usage due to VM management. The current system creates a number of VMs equal to the number of threads for each SC to allow for performant intra-contract parallelization when commutativity support is enabled. Although performant, it suffers from high RAM usage, which may be wasteful in cases where the blockchain lacks hot-spot workloads.
- Reliance on perfect RWS. The system relies on the assumption that either the RWS is given, or that the predicted RWS by parsing the contract's profile is always accurate. This limits the system capability to work with transactions whose execution path depends on some storage item's value.

To enhance this work, some key points could be improved in **future work**:

- Improve the schedule creation & validation via light parallelization strategies. Note that a parallel schedule creation strategy was already proposed, but as explained, it provided slightly higher times than the serial version, mostly due to the merging step, as well as for the conversion from the sequential to the parallel schedule version, which must have some of its structures wrapped around locks to ensure correctness. This overhead is most probably due to implementation inefficiencies, and further optimizations at this front could provide improved performance.
- Improve VM management. A lock-free stack-based approach could provide as a beneficial tradeoff between RAM usage and performance. The idea is that threads would push and pop VMs in a FILO order. This would make threads use the VMs following a Last Recently Used (LRU) eviction policy, i.e., threads would always try to pop the same first VM. This would respect the principle of locality as highly-common accessed VMs would be more susceptible to be stored in cache, whereas a with a queue, each thread would pop some random VM following the order they were inserted in the queue.
- Improve queue management. Both ready and partial ready queues could use a batch pop instead of popping one transaction at a time. Currently, queues do not respect FIFO order, as they do not need to, and are implemented using concurrent Hash Sets. This is because we need to be able to remove some transaction from `QueuePartialReady` by the transaction ID. A custom data structure that allows for efficient pushing and concurrent batched popping as well as removal by value could provide beneficial.

- Improve commutative operation's representation at operation nodes. As explained earlier, commutative operations currently incur some overhead as they are deserialized, merged and then serialized again. A better approach would be to deserialize the base storage value only once for each item, and for each commutative operation, we would use the in-memory value for the merging, and for storing at the node. Only when encountering a non-commutative write or when persisting the schedule we would serialize the merged value.

- Support incomplete and inaccurate RWS predictions. As pointed earlier, the system currently assumes a perfect RWS prediction. One approach to further extend the system's capabilities would be to support identification of untracked operations at runtime, and abort transactions according to the new dependencies, as proposed by Qi et. al. [3]. Another way of solving this issue would be to employ some sort of OCC strategy for transactions with incomplete or inaccurate RWSs, and employ scheduled execution only for the transactions with perfectly predictable RWS. Nevertheless, the system already is equipped with some base support for such improvements. As discussed in the Profile Generation section in Chapter 4, the current `TxRWS` structure already comes with a `profile_status` field, indicating whether the current profile is complete or not. This can be used to identify transactions with an incomplete RWS prediction, and decide how the transaction execution is to be handled. Furthermore, we also include the field `storage_dependency` to identify if, when parsing the profile, we encountered any condition dependent on storage. This provides information about the RWS, marking it as a possibly wrong prediction, but complete nonetheless. This allows the system to handle execution of such transactions accordingly.

Bibliography

[1] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, “Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2315–2326.

[2] H. Lin, Y. Zhou, and L. Wu, “Operation-level concurrent transaction execution for blockchains,” *arXiv preprint arXiv:2211.07911*, 2022.

[3] X. Qi, J. Jiao, and Y. Li, “Smart contract parallel execution with fine-grained state accesses,” in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2023, pp. 841–852.

[4] S. Haber and W. S. Stornetta, *How to time-stamp a digital document*. springer, 1991.

[5] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[6] V. Buterin *et al.*, “Ethereum white paper,” *GitHub repository*, pp. 22–23, 2013.

[7] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[8] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[9] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, 2013.

[10] J. Kwon and E. Buchman, “Cosmos whitepaper,” *A Netw. Distrib. Ledgers*, vol. 27, pp. 1–32, 2019.

[11] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, “An overview of blockchain technology: Architecture, consensus, and future trends,” in *2017 IEEE international congress on big data (BigData congress)*. Ieee, 2017, pp. 557–564.

[12] R. Amir. Understanding an ethereum transaction. Accessed October 12, 2024. [Online]. Available: https://info.etherscan.com/understanding-an-ethereum-transaction/

[13] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *compsurv*, vol. 51, no. 3, 2018.

[14] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. 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*, 2023, pp. 232–244.

[15] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "Optsmart: a space efficient opt imistic concurrent execution of smart contracts," *Distributed and Parallel Databases*, pp. 1–53, 2022.

[16] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 303–312.

[17] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," *arXiv preprint arXiv:1901.01376*, 2019.

[18] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[19] ethereum.org. Merkle patricia trie. Accessed October 12, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/

[20] solidity.org. Merkle patricia trie. Accessed October 12, 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.6.1/

[21] ibcprotocol.dev. Ibc protocol. Accessed October 9, 2024. [Online]. Available: https://www.ibcprotocol.dev/

[22] J. Kwon, "Tendermint: Consensus without mining," *Draft v. 0.6, fall*, vol. 1, no. 11, pp. 1–11, 2014.

[23] cosmos.org. Application-specific blockchains. Accessed October 12, 2024. [Online]. Available: https://docs.cosmos.network/main/learn/intro/why-app-specific

[24] ——. Cosmos sdk application anatomy. Accessed October 12, 2024. [Online]. Available: https://docs.cosmos.network/main/learn/beginner/app-anatomy

[25] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on Computer architecture*, 1993, pp. 289–300.

[26] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, “An effective hybrid transactional memory system with strong isolation guarantees,” in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 69–80.

[27] J. M. Faleiro and D. J. Abadi, “Rethinking serializable multiversion concurrency control,” *arXiv preprint arXiv:1412.2324*, 2014.

[28] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, “Calvin: fast distributed transactions for partitioned database systems,” in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 1–12.

[29] Y. Li, H. Liu, J. Gao, J. Zhang, Z. Guan, and Z. Chen, “Accelerating block lifecycle on blockchain via hardware transactional memory,” *Journal of Parallel and Distributed Computing*, vol. 184, p. 104779, 2024.

[30] M. J. Amiri, D. Agrawal, and A. El Abbadi, “Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1337–1347.

[31] H. Li, Y. Chen, X. Shi, X. Bai, N. Mo, W. Li, R. Guo, Z. Wang, and Y. Sun, “Fisco-bcos: An enterprise-grade permissioned blockchain system with high-performance,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–17.

[32] D. Xian and X. Wei, “Sc-chef: Turboboosting smart contract concurrent execution for high contention workloads via chopping transactions,” *IEEE Transactions on Reliability*, 2023.