Concerto: Transaction-Parallel EVM

Stephen Su*
Stanford University
stephensu@cs.stanford.edu

Jeffrey Hu*
Stanford University
jeffhu@cs.stanford.edu

Yashodhar Govil*
Stanford University
ygovil@stanford.edu

Sumer Kohli*
Stanford University
sumer@cs.stanford.edu

Abstract

One major hurdle in the path to greater blockchain and cryptocurrency adoption is the limitation on throughput for sequential execution engines. To address this restriction, we propose Concerto, a parallel execution engine for the Ethereum Virtual Machine (EVM). Concerto aims to exploit the inherent parallelism of blockchain transactions, particularly those that do not share state dependencies. By grouping transactions with no related dependencies, and using a low overhead collaborative scheduler, each block's transactions can be processed in parallel without violating the blockchain's deterministic execution requirements. Our implementation is in Rust and is based on Block-STM [10]. Given different SSD read latencies, Concerto provides a 1.1 to 4x speedup with 8 threads on a modern machine, which we demonstrate approaches the theoretical maximum speedup for this application.

1 Introduction

Blockchains maintain a trustable distributed ledger and are structured as deterministic replicated state machines. Bitcoin [13], the first blockchain, allowed for financial infrastructure to be done without a central authority. Since Bitcoin, other blockchains, including notably Ethereum [16], have been created. The main contribution of Ethereum is introducing a Turing-complete execution platform, called the Ethereum Virtual Machine (EVM), which loads resources and executes scripts known as smart contracts through transactions. Smart contracts have since appeared in other blockchains, and allow for applications such as decentralized finance (DeFi) [6], non-fungible tokens (NFTs) [15], banking [2], and trade [5]. Ethereum has become a dominant player in the blockchain space and as of June 2024, there is more than 192 billion USD were locked in the DeFi ecosystem, with Ethereum making up 130 billion USD, or 68%, of the total [8].

The throughput of the most popular blockchains remain very low, especially relative to traditional payment systems such as PayPal or Visa. In practice, Bitcoin maintains 6 to 8 transactions per second (TPS), and Ethereum has 12 to 15 TPS [4]. There is increased attention to improving throughput as blockchain usage increases. While progress has been made for scaling different parts of the system, performance is still bottlenecked by transaction execution.

In blockchains, transactions are grouped in blocks, and an execution of a block should always result in the same deterministic outcome. Most blockchains have transactions executed sequentially, where the results of one transaction are calculated before executing the next transaction. However, transactions that access separate memory locations can be executed in parallel. Note that blockchain transactions have the potential to have a significant number of memory access conflicts due to frequent access of smart contracts or arbitrage opportunities. Nonetheless, blocks produced in practice have enough parallelizable transactions that parallel execution can still be used to improve performance. Transaction parallelization was previously explored with Block-STM, but Block-STM was originally implemented for the Aptos blockchain, which runs on the Move virtual machine.

In this paper, we present Concerto, a parallel execution engine for smart contracts on the EVM. The result of an execution by Concerto is the same as the sequential execution in the preset order that they appear in the block. We evaluate Block-STM on live blocks from Ethereum.

Contributions: In summary, we make the following contributions:

- We present Concerto, an implementation of transaction parallelization on EVM in Rust.
- We analyze the practical efficacy of parallel execution on live-EVM blocks and characterize the improvements from parallelization depending on IO delays.
- We quantify what improvements can be theoretically expected given a transaction dependency graph.

2 Related Work

Detecting and working with conflicting transactions is the main challenge in this setting. One approach to this is studied in the Software Transaction Memory (STM) libraries [11, 14], where memory accesses are tracked to detect conflicts. There are also some STM libraries that implement optimistic concurrency control [9] (OCC) where memory accesses are recorded, transactions are validated post execution, and conflicting transactions are aborted and re-executed. In general, the blockchain setting requires a deterministic setting which many STM papers do not achieve. Other works use specific parts of the blockchain in conjunction with STM by precomputing dependencies and calculating a directed acyclic graph of transactions which can be executed using a fork-join schedule [3]. There have also been works related to accelerating Ethereum execution, such as Forerunner [7], which gains performance by pre-executing transactions to give hints for final execution.

3 Design

3.1 System Overview

Broadly, Concerto implements the Block-STM concurrency control approach around an EVM. To achieve this, the parallelization approach taken is centered around a pool of workers that pick off execution and validation tasks from a scheduler data structure. Transactions are executed optimistically, and the database is instrumented to track the read and write sets that occur during execution. At validation time, each worker checks whether a given state access has been invalidated. If all previous transactions have been both committed and the current transaction has been both executed and validated, then the current transaction can be committed as well.

3.2 Data Partitioning

There are several different data structures with different levels of visibility. Across all threads, the main data structures are a multi-view hashmap (MVHashMap) which tracks for each transaction / state tuple and a read-only snapshot of the pre-block state. On an individual thread basis, each thread creates an instrumented database per execution. This database is a pass through type to other views - namely the MVHashMap and state snapshot. Finally, all threads share a transaction input-output object, which tracks the read and write sets of all transactions.

During an execution of a transaction, each thread instantiates an individual virtual machine. The main access is through the instrumented database. For a given state access, the instrumented database first marks that state slot as touched internally. Then, the MVHashMap is checked to see whether a prior transaction has written to the current transaction. If it has, then the result is immediately returned, otherwise the state snapshot is queried and returned. After execution is finished, the collected reads and writes are taken from the instrumented database and stored in the input-output objects.

At validation time, the readset for a given transaction is pulled from the global input-output object and compared against the latest reads and writes contained in the MVHashMap. If there has been a more recent write to that slot compared to the one accessed by the transaction, validation fails. Note that validation success does not imply that the transaction is committed, as validation failure of an earlier transaction requires failure of all downstream transactions (new state could be written to invalidating later execution).

3.3 Scheduling

The main innovation that Block-STM employs is using atomic counters rather than locks to reduce synchronization overhead. Concerto leverages this by using an atomic counter for both execution indices. Each worker runs in a loop, querying the scheduler for the latest task each iteration.

Another optimization that is included is that failed transactions are not rescheduled for execution until the parent transaction that failed is updated. Instead, that particular state slot is marked as an ESTIMATE and all transactions that depend on it wait on a condition

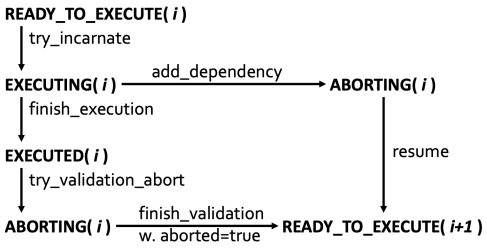

Figure 1: Lifecycle of a transaction from Gelashvili et al.

variable that is updated after the original transaction is successfully validated.

3.4 Error Handling

Because the system is executing speculatively, there are many cases where faults can occur before or during execution. In Ethereum accounts, for instance, nonces are incremented as a way to prevent replay attacks. However, this leads to `NonceTooLow` / `NonceTooHigh` errors through an execution. This is handled by setting a speculative execution flag inside the captured reads object. Then, at validation time, the system notes that a speculative execution failure has occurred and fails the validation of the transaction.

4 Implementation

We implemented Concerto in ≈4K lines of Rust code using `rustc` 1.78.0, based on Block-STM [10]. Our implementation supports retrieving transaction blocks from Alchemy [1], caching them locally, and executing them serially or in parallel with a specified concurrency limit. Our codebase, workflow, and associated documentation is publicly available at GitHub.

5 Evaluation

5.1 Empirical Speedup

Figure 2 demonstrates the significant performance benefits of Concerto from parallelization, measuring throughput of a block execution over 250 trials for 1 to 8 threads and an SSD delay from 0 to 50 μs. All measurements were taken on a 2022 MacBook Pro with an M1 Pro processor featuring 8 performance cores, hence the upper bound of 8 threads. To ensure a fair comparison with sequential execution, when evaluating with one thread, we performed a 'vanilla' sequential execution without any scheduling or bookkeeping overhead (*e.g.,* MVHashMap's or similar). SSD delays are added and ablated against in Figure 2 because our implementation maintains the transactions entirely in memory, which is an unrealistic model of EVM execution — typical SSD delays range from 10 to 50 μs [12]. Increasing the SSD delay not only directly dampens the throughput (TPS), but also denoises the throughput by dominating delays from scheduling, cache behavior, and optimistic concurrency control re-executions.

For an SSD delay of 50 μs, increasing the number of threads leads to 8 leads to a **4x** improvement in transactions per second (TPS) when compared to a sequential execution. Even an SSD delay of 5 μs demonstrates a more than **2x** throughput at 8 threads relative to sequential execution. Removing SSD delays entirely shows a negligible increase in throughput as parallelism increases due to transactions taking nearly no time at all to complete (no disk accesses needed, so just state operations on memory). However, no SSD delays is an unrealistic execution model; we only include that curve for completeness. These findings emphasize the potential of transaction parallelization in enhancing the scalability and performance of the Ethereum Virtual Machine.

5.2 Theoretical Speedup Limits

To better anchor our evaluation, we consider the 'theoretical limit of speedup'. This limit can help us understand the fastest rate at which a blockchain can operate if we optimize it for parallel processing. This limit is important because it represents the peak performance we can aim for. By knowing this, we can measure how much improvement is possible from our current methods and identify the most effective ways to increase the speed of transactions. This section of the paper will explain these ideas in simple terms, providing a clear target for our technological advancements and a benchmark against which to compare our progress.

To approximate this limit, we first capture the read and write sets of all transactions within a block. These sets are crucial as they determine which transactions can be executed in parallel without causing conflicts. Next, we use these sets to construct a dependency graph. This graph represents the transactions as nodes, with edges indicating dependencies based on their read and

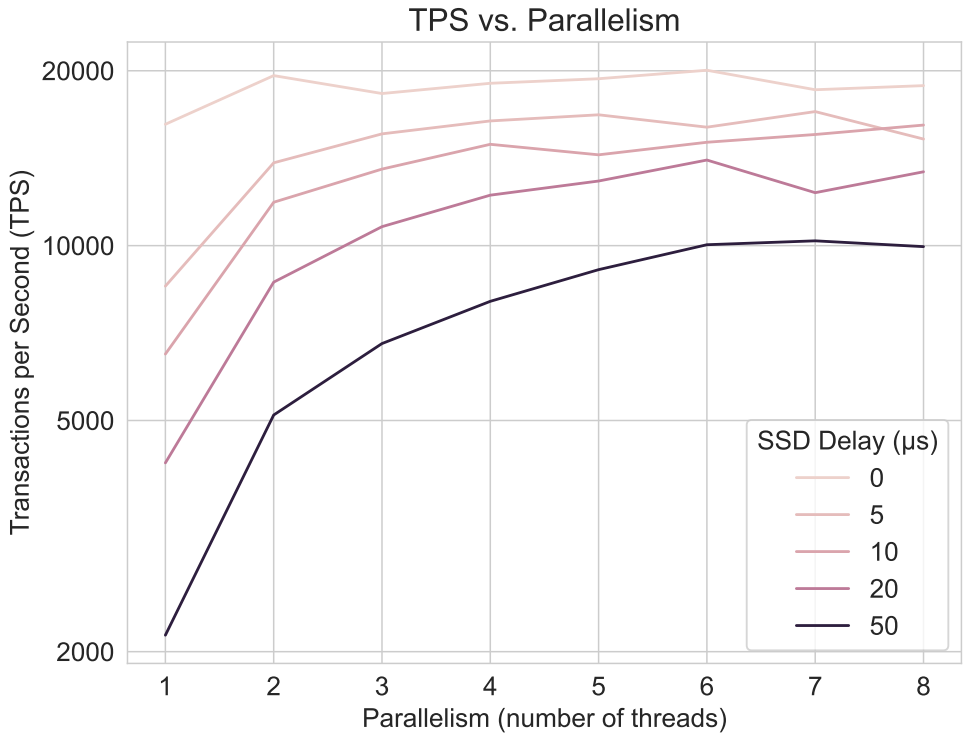

Figure 2: Transactions per second (TPS) vs parallelism (in number of threads), broken out by SSD read delay for each storage key. Each data point represents the average of 250 trials for 1 to 8 threads and 0, 5, 10, 20, and 50 μs SSD delays. The y-axis is log-scale and starts at 2000 TPS.

write sets. Transactions without interdependencies can potentially be processed in parallel, leading to increased throughput. Finally, we run simulations using a simple execution engine. Our execution engine includes a penalty in the event that a transaction must be rolled back due to an incomplete dependency. These simulations allow us to experiment with different transaction execution times and scheduling policies, providing insights into how various factors impact the overall processing speed. By adjusting these parameters, we can identify optimal strategies for parallel transaction execution, allowing us to approximate a theoretical limit of speedup.

We created simulations using the following execution time policies:

(1) **Constant Time:** Each node (representing one transaction) takes a constant amount of time to execute. In our simulation, this constant was set to 1.0ms.
(2) **Random Time:** Each node takes a uniformly random amount of time to execute. In our simulation, the random number is sampled from the range [0.25, 1.5]ms.

and scheduling policies:

(1) **Uniform Random:** Selects a node at random (representing one transaction) that was not just rolled back to assign to the next worker. If the node cannot be run due to incomplete dependencies, the thread incurs a rollback cost of 0.5ms.
(2) **Greedy Oracle:** Selects a node to schedule by maximizing the depth of its connected component (or width to tie-break). The depth of a node is the length of the longest path to a leaf node in its component, and the width is the largest topological "generation". In other words, the algorithm selects first for the node with the longest, and then widest, chain of dependencies. This policy only selects nodes from those that have no

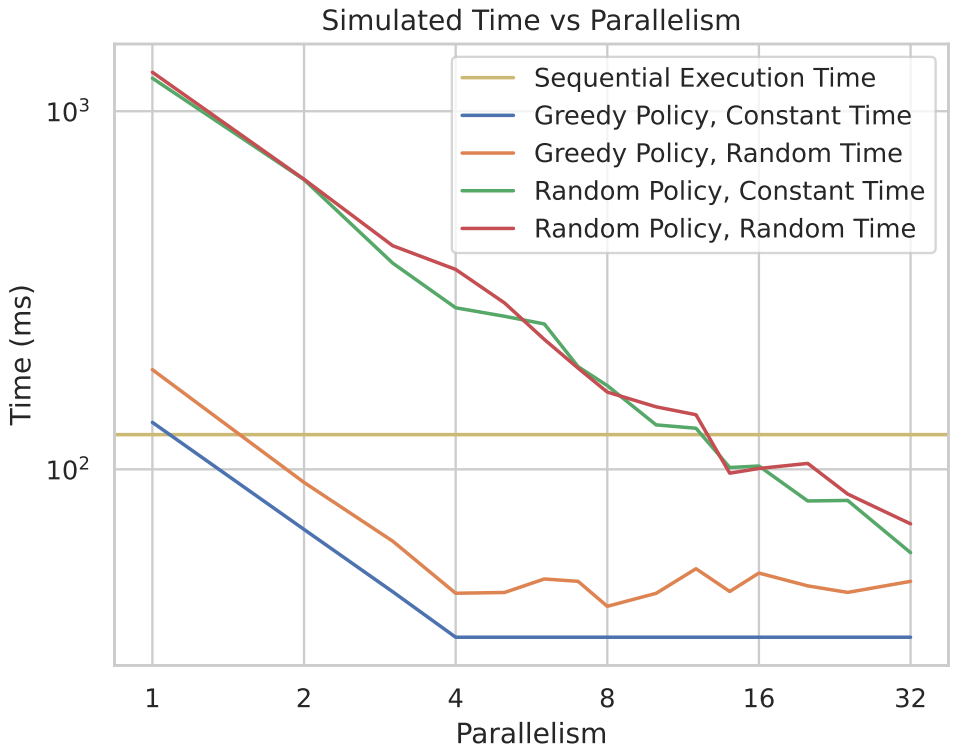

Figure 3: Time Taken (ms) vs Parallelism (in number of threads) in simulation, broken out by different scheduling and execution time policies. The x-axis and y-axis are both log-scale. The yellow bar represents parity with sequential execution time.

incomplete dependencies, so no rollback costs are incurred.

Figure 3 demonstrates the expected theoretical speedup under combinations of different scheduling and execution time policies. We display a yellow horizontal bar at the threshold of parity with sequential execution — hence, of interest is the parallelism factor at which each curve intersects the yellow bar. We note that the Greedy Oracle policy with a constant time is the first to intersect, unsurprisingly, as it can operate with perfect information and zero noise. Both baseline policy simulations, with a Uniform Random policy and either constant or random execution times, are very slow to approach the threshold of improvement, illustrating the importance of the scheduling policy in improving throughput. Our empirical performance in Figure 2 more closely resembles that of the Greedy Oracle policy, and increases in resemblance as the SSD delay increases. Hence, we are able to achieve essentially optimal performance given our scheduling policy, as well as the desired improvement of about **4x**, which is the maximum theoretical improvement (*i.e.,* the blue curve plateaus at 4x improvement, which is the number of transactions in the block divided by the length of the longest dependency chain).

6 Discussion

Implications. The implications of our work on Concerto are twofold. Practically, the performance improvements achieved through parallel transaction execution empower Ethereum and EVM-based systems to handle higher transaction volumes, reduce confirmation times, and enhance user experience, driving mainstream blockchain adoption. From a research perspective, our findings contribute to advancing parallel execution engines for blockchain systems, serving as a valuable reference for

comparison with state-of-the-art projects. By demonstrating the viability and benefits of transaction parallelization, our work inspires further innovation in the field.

Limitations. Concerto, while demonstrating significant performance improvements through parallel transaction execution, has limitations. As a research proof of concept, further optimizations and robustness enhancements are necessary for production deployment. High contention workloads may limit performance gains or even lead to worse performance compared to sequential execution due to coordination overhead. Moreover, Concerto's performance boost relies on the underlying database's ability to efficiently support asynchronous operations, without which the full potential of the parallel execution engine may not be realized.

Future Work. Future work on Concerto can explore several avenues to enhance the performance and practicality of parallel transaction execution in blockchain systems. Optimizing the underlying database to support asynchronous operations and parallel processing could unlock the full potential of Concerto's parallel execution engine. Investigating static analysis techniques to identify transaction dependencies and optimize the scheduling of parallel execution could lead to further performance improvements. Conducting extensive testing and evaluation on larger-scale blockchain networks with diverse transaction types and realistic workloads would provide valuable insights into its performance and scalability. Additionally, exploring the integration of Concerto with existing blockchain platforms and tools would facilitate its adoption and usability. Addressing these aspects can contribute to the development of a more robust, efficient, and production-ready parallel transaction execution engine for blockchain systems.

References

[1] [n. d.]. Alchemy - the web3 development platform. https://www.alchemy.com/.

[2] Abdullah Al Mamun, Sheikh Riad Hasan, Md Salahuddin Bhuiyan, M Shamim Kaiser, and Mohammad Abu Yousuf. 2020. Secure and transparent KYC for banking system using IPFS and blockchain technology. In *2020 IEEE region 10 symposium (TENSYMP)*. IEEE, 348–351.

[3] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2022. OptSmart: a space efficient Opt imistic concurrent execution of Smart contracts. *Distributed and Parallel Databases* (2022), 1–53.

[4] Blockchair. 2024. Ethereum Transactions Per Second Chart. https://blockchair.com/ethereum/charts/transactions-per-second Accessed: 2024-06-07.

[5] Yanling Chang, Eleftherios Iakovou, and Weidong Shi. 2020. Blockchain in global supply chains and cross border trade: a critical synthesis of the state-of-the-art, challenges and opportunities. *International Journal of Production Research* 58, 7 (2020), 2082–2099.

[6] Yan Chen and Cristiano Bellavitis. 2020. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights* 13 (2020), e00151.

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

[8] Christopher Roark. 2024. DeFi TVL Reaches $192B for First Time in 15 Months — DappRadar. (2024). https://cointelegraph.com/news/defi-tvl-192-billion-first-time-15-months-dappradar Accessed: 2024-06-07.

[9] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.

[10] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. 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*. 232–244.

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

[12] Intel. [n. d.]. Achieve Consistent Low Latency for Your Storage-Intensive Workloads. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/low-latency-for-storage-intensive-workloads-tech-brief.pdf.

[13] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[14] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. 204–213.

[15] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447* (2021).

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