FAFO: Over 1 million TPS on a single node running EVM while still Merkleizing every block

Ryan Zarick Isaac Zhang Daniel Wong Thomas Kim Bryan Pellegrino
Mignon Li Kelvin Wong

LayerZero Labs Ltd

Abstract

Current blockchain execution throughput is limited by data contention, reducing execution layer parallelism. Fast Ahead-of-Formation Optimization (FAFO) is the first blockchain transaction scheduler to address this problem by reordering transactions *before block formation* for maximum concurrency. FAFO uses CPU-optimized cache-friendly Bloom filters to efficiently detect conflicts and schedule parallel transaction execution at high throughput and low overhead.

We integrate the Rust EVM client (REVM) into FAFO and achieve over 1.1 million native ETH transfers per second and over half a million ERC20 transfers per second on a single node (Table 1), with 91% lower cost compared to state-of-the-art sharded execution. Unlike many other existing high throughput blockchain execution clients, FAFO uses QMDB to Merkleize world state after *every block*, enabling light clients and stateless validation for ZK-based vApps.

FAFO scales with minimal synchronization overhead, scaling linearly with additional CPU resources until it fully exploits the maximum parallelism of the underlying transaction flow. FAFO proves that the high throughput necessary to support future decentralized applications can be achieved with a streamlined execution layer and innovations in blockchain transaction scheduler design. FAFO is open-sourced at `https://github.com/LayerZero-Labs/fafo`.

1 Introduction

Blockchain execution has historically traded off throughput and decentralization; higher throughput always meant higher capital requirements, reducing decentralization. In theory this is asymptotically true, but most blockchains are at least partially bottlenecked by the grossly inefficient use of modern multi-core CPUs on the execution layer. The goal of our work is to efficiently utilize multi-core CPUs to achieve high throughput at low cost without introducing the complexities of sharding.

At a high level, blockchain execution begins with a block *producer* selecting transactions from the mempool and packing them into blocks; these blocks are executed against the blockchain state transition function (STF) by one or more nodes in a decentralized network. In most blockchains, transactions are packed into blocks with little to no regard to parallel execution, and the execution client either runs these transactions in serial or tries to parallelize a potentially unparallelizable transaction flow. As a result, the execution layer makes very inefficient use of modern multi-core CPUs, requiring expensive overprovisioning or sharding to achieve high transactions per second (TPS).

Blockchain transactions must achieve *deterministic serializability*, meaning that the schedule of all transactions across all blocks must be equivalent to a deterministically chosen total ordering of those transactions. In this paper, we focus on parallelizing execution of EVM [20] transactions, but the techniques we leverage are applicable to most blockchain VMs. We recognize that transaction serializability is achievable if the schedule of all *storage operations* is *conflict-serializable* [8], and that this property is sufficient to implement transactional semantics for most if not all blockchain VMs. We take advantage of this in our end-to-end scheduling and execution runtime, Fast Ahead-of-Formation Optimization (*FAFO*), which identifies and exploits transaction-level parallelism through heuristic transaction scheduling, dispatch, and execution.

FAFO not only exceeds the state-of-the-art for single-node throughput, but also *Merkleizes every block*; this enables light clients and stateless validation which is required for Zero-Knowledge (ZK)-based vApps [22]. Many other blockchains [21, 19] Merkleize at very

Benchmark	Throughput (TPS)
Native	1,121,732
ERC20	565,956

Table 1: FAFO achieves up to 1.1 M transactions per second (TPS) on a 96-core server using a synthetic benchmark (§ 3.2)

low frequency or forego Merkleization altogether to sidestep the storage bottleneck; FAFO instead leverages the QMDB verifiable database [23] to Merkleize at high throughput with no compromises.

Our contributions in this paper are threefold: first, a novel CPU-efficient, cache-friendly data structure (*ParaBloom*) to detect data conflicts. Second, an efficient algorithm (*ParaFramer*) that uses *ParaBloom* to generate a low-contention, highly parallelizable transaction stream. Third, we design an efficient scheduler (*ParaScheduler*) that schedules conflict-serializable parallel execution of the transaction stream. These three components are combined into an efficient end-to-end scheduler and execution pipeline (FAFO), which we demonstrate can process over 1.1 million TPS and Merkleize every block *on a single node* (Table 1) running EVM transactions on REVM backed by QMDB.

2 Design

In this section, we present the design of FAFO in detail and argue its correctness and optimality. We define conflict-serializability and transaction-level parallelism in the context of blockchain execution (§2.1). In §2.2, we present a high-level overview of the four-stage pipelined design of FAFO. We follow this with a detailed description of each component: *ParaLyze* (§2.3), *ParaFramer* & *ParaBloom* (§2.4), and *ParaScheduler* (§2.5).

2.1 Transaction-level parallelism

Two transactions *conflict* if they both access the same data and at least one access is a write, and a *conflict-serializable* transaction schedule S' is such that there exists a serial schedule S where the result of all conflicting memory accesses is equal between S and S' [8].

Parallelism exists on a *per-transaction* granularity: transactions can run concurrently as long as their storage accesses are conflict-serializable. To capture this idea, we define Transaction-Level Parallelism (TLP) as the average number of transactions that can be run concurrently given some scheduling of a set of transactions. There exists a theoretical upper bound on TLP for any set of transactions, and we refer to the *optimal schedule* as any transaction schedule that achieves this upper bound.

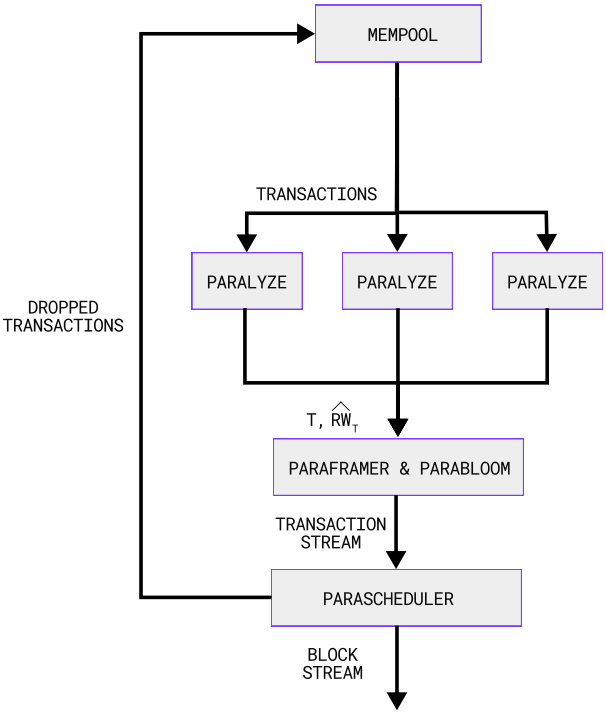

Figure 1: **System Architecture.** Concurrent instances of *ParaLyze* preprocesses transactions, *ParaFramer* packs frames using *ParaBloom*, and *ParaScheduler* produces the final transaction execution ordering and defines block boundaries.

The optimal schedule can be constructed via an optimal coloring of a precedence graph across all transactions, and real-world performance is maximized if the size of each color class is roughly equal to the number of execution units (cores) of the execution client. Unfortunately, it is impractical to compute the globally optimal coloring for two reasons. First, the computational cost of constructing and coloring the graph is high. Second, the streaming nature of the workload demands some bound on service latency, limiting the number of unexecuted transactions that can be accumulated in the mempool. FAFO strikes a balance between theory and practice, extracting the majority of parallelism from the workload without degrading end-to-end latency or overusing compute resources.

2.2 FAFO Architecture

We assume that block producer election occurs on a reasonably long timescale (e.g., EOS [3], Solana[21]) to increase the maximum TLP of the pool of transactions available to a single block producer. FAFO is compatible with a wide range of blockchain applications such as vApps [22], rollups, and L1 chains. In our model, the

block producer produces a single totally-ordered stream of transactions $\{T_0, T_1, \ldots, T_i\}$ and inserts block headers into the stream to mark block boundaries. Without loss of generality, we assume that the underlying verifiable database (e.g., QMDB [23]) is not a bottleneck of the system.

FAFO processes transactions in a four-stage pipeline:

1. ***ParaLyze*** preprocesses each transaction in the mempool to approximate the addresses of every storage slot it reads and writes. This approximation is *not* guaranteed to accurately reflect the state accessed during execution, as the ordering of transactions is not final at this stage.

2. ***ParaFramer*** identifies non-conflicting transactions by computing the intersection of their approximate read/write sets using *ParaBloom* (§2.4). It then packs these non-conflicting transactions into *frames*, and outputs a stream of transactions grouped by frame.

3. ***ParaScheduler*** dispatches and executes the stream of transactions received from *ParaFramer*, identifying and exploiting any transaction-level parallelism in the process. Transactions whose actual read/write set diverge from their approximate read/write sets are dropped and returned to the mempool.

4. **Block formation**: FAFO periodically synchronizes the execution threads, flushes storage, and inserts block headers into the stream of non-dropped transactions output by *ParaScheduler*. These blocks can then be settled to the underlying consensus layer.

2.3 *ParaLyze*: Transaction Static Analysis

ParaLyze enforces that all transactions in the mempool are well-formed and annotated with the set storage reads and writes that they will perform. We term this the *approximate read/write set* and denote the approximate read/write set of a transaction T_i as $\widehat{RW}_{T_i} = \{\widehat{R}_{T_i}, \widehat{W}_{T_i}\}$. $\widehat{RW}_{T_i}$ is determined either by metadata (e.g., full node-provided access list [4]) or simulation; without loss of generality, we assume the latter.

ParaLyze does not persist any writes during simulation to avoid bottlenecking this stage on storage contention; as a result, many transactions are simulated against the same initial state which may diverge (in a conflicting manner) from the state during actual execution. We use distributed parallel simulation to maximize throughput, and accept that approximate read/write sets may sometimes be inaccurately captured for workloads such as data-dependent flows or those which rely heavily on modifying global state.

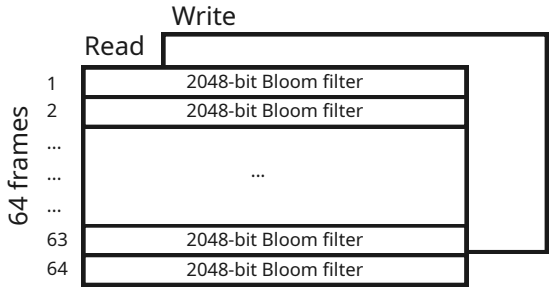

Figure 2: ***ParaBloom* Layout.** Each active frame stores two 2048-bit Bloom filters (read and write).

2.4 *ParaFramer* & *ParaBloom*: Conflict detection and frame packing

ParaBloom identifies and groups non-conflicting transactions into *frames*, and *ParaFramer* forms the transaction stream from these frames. This transaction stream is *parallelism-aware*, with average TLP greater than or equal to the average number of transactions per frame.

We implement *ParaBloom* using a collection of Bloom filters to efficiently detect data conflicts. *ParaBloom* compactly represents the union of the read and write sets of all transactions in each frame using two Bloom filters, one for read and one for write. This allows efficient conflict checks via CPU-efficient bitwise operations.

ParaFramer does not generate a *theoretically* optimal transaction stream due to the two impracticalities mentioned at the end of §2.1. First, the Bloom filters in *ParaBloom* have some probability of falsely identifying data conflicts; this reduces TLP by about 8% in our testing but also significantly improves the end-to-end throughput and efficiency of FAFO. Second, we do not perform an exhaustive search of all combinations of transactions in the mempool, but instead greedily pack them in a single streaming pass over the mempool.

The computational complexity of conflict-serializable scheduling is generally linear in the size of the precedence graph [5], so a high degree of contention increases the number of edges and, by extension, the computational cost of scheduling. Our frame-based approach uses lightweight cache-friendly data structures and approximation to amortize the cost of checking conflicts and reduce the number of checks respectively.

***ParaBloom*:** For transaction T packed in frame F_i, we store $\widehat{RW}_T$ in *ParaBloom* for frame F (PB_F). Each frame in *ParaBloom* is represented by two Bloom filters: one for the frame's *aggregate read set* ($\widehat{AR}_F$), and another for its *aggregate write set* ($\widehat{AW}_F$) (Figure 2). Transaction T

is *admissible* to frame F_i iff no storage operation conflicts between F_i and T, or more formally:

$$Admissible(T, \widehat{RW}_T, \widehat{PB}_F): \\ (\widehat{R}_T \cap \widehat{AW}_F) = (\widehat{W}_T \cap \widehat{AR}_F) = (\widehat{W}_T \cap \widehat{AW}_F) = \varnothing \quad (1)$$

Algorithm 1 Greedy Frame-Packing Algorithm

```
1:  Input: Transaction list T, read/write sets RW_{T_i}
2:  Output: Frames (sets of non-conflicting txns)
3:  Initialize F empty frames
4:  for T_j in T do
5:      placed ← false
6:      for each frame F_i in Frames do
7:          if Admissible(t, RW_{T_i}, PB_{F_i}) (Eq. (1)) then
8:              txns_i ← txns_i ∪ {t}
9:              ÂR_{F_i} ← ÂR_{F_i} ∪ R̂_{T_j}
10:             ÂW_{F_i} ← ÂW_{F_i} ∪ Ŵ_{t_j}
11:             placed ← true; break
12:         end if
13:     end for
14:     if placed = false then
15:         f_x ← largest frame
16:         Finalize(f_x)
17:         f_x ← {T_j}
18:         ÂR_{T_j} ← R̂_{T_j}
19:         ÂW_{T_j} ← Ŵ_{T_j}
20:     end if
21: end for
```

***ParaBloom* Frame Packing:** *ParaBloom* packs frames greedily, placing each transaction into the lowest index non-conflicting frame in *ParaBloom*. If the transaction conflicts with all frames in the active set, *ParaBloom* finalizes (commit) the largest existing frame, replaces it with a new empty frame, then inserts the new transaction into the empty frame (illustrated in Algorithm 1). This design encourages large average frame sizes, directly correlated with a better lower bound of TLP. These ejected frames are returned to *ParaFramer* which then sends them to *ParaScheduler*. Optionally, *ParaFramer* can define a secondary policy (e.g., number of transactions, time) to eject frames from *ParaBloom*.

ParaBloom has two parameters: the number and size of each Bloom filter. Our testbed uses a 64-bit CPU with 64 kB of L1 cache, so we configure *ParaBloom* to use 64 pairs of 2048-bit Bloom filters; this allows *ParaBloom* to index frames using a bitmap stored in a single 64-bit word and fit neatly into 32 KiB (half of L1 cache). We recommend parameterizing the number of frames based on the word size of the CPU, then sizing each Bloom filter with the maximum number of bits that will keep the entire structure within half of L1 cache.

2.5 *ParaScheduler*: Transaction Scheduling and Execution

ParaScheduler begins by reconstructing the same frames generated by *ParaFramer* from the transaction stream. As frames are ejected from the active set, we insert each of its transactions T_i into a collection of directed acyclic graphs (DAGs). For each storage slot s in RW_{T_i}, there exists a corresponding DAG; the transaction is inserted into the DAG and a directed edge is added from each transaction T_j that conflicts with T_i such that $(T_j \rightarrow T_i \wedge \neg\exists T_k | T_j \rightarrow T_k \rightarrow T_i, j < k$ where $\rightarrow$ is the happens-before relation). Each of the DAGs is constructed by a separate thread, and executed transactions are pruned from the DAG to reduce the iteration space. Furthermore, T_i does not need to be checked against any other transactions in the same frame, as by construction there are no data conflicts.

ParaScheduler dispatches a transaction immediately after all ancestors have completed execution to ensure that transactions are safely dispatched and executed as soon as possible. After execution, the actual read/write set of the transaction T_i (RW_{T_i}) is recorded and compared to $\widehat{RW}_{T_i}$. If $RW_{T_i} \neq \widehat{RW}_{T_i}$, T_i is dropped from the stream to preserve conflict-serializability, then returned to the mempool to be rescheduled; we expect this to be rare in practice. Intuitively, the TLP of the transaction schedule executed by *ParaScheduler* will be equal to or greater than $\frac{transactions}{frames}$, since *ParaScheduler* can at least execute all transactions in each frame in parallel.

We argue that this collection of DAGs is equivalent to a precedence graph, and that the schedule produced by this DAG achieves equal or greater TLP than executing the transactions frame-by-frame. Without loss of generality, we assume an unbounded number of CPU cores on the server running *ParaScheduler*. Suppose only 1 storage slot is ever accessed: by construction, the DAG produced by *ParaScheduler* is equivalent to the precedence graph of the operations. Additionally, the total ordering of the stream ensures this DAG is a forest of directed trees. Under the same assumption, there must exist no edge between any two transactions in the same frame; *ParaScheduler* only delays execution of any given transaction T_i if its direct ancestor T_j has not yet been executed, and T_i must have no direct ancestor within the same frame.

Relaxing the assumption of a single storage slot, the "schedule" S generated by the DAG for each storage slot a (denoted S_a) is conflict-serializable for that particular slot. Each transaction is executed only after the schedule

S_a permits, so it is impossible for the executed schedule to have any conflicts for slot a. For a collection of DAGs, *ParaScheduler* waits until all schedules S_{a_x} permit the execution of transaction T_i before dispatching T_i, guaranteeing that the schedule of operations in T_i is conflict-serializable across all storage slots S_{a_x}.

Summary FAFO reorders transactions to maximize TLP *ahead of* block formation–this design choice enables FAFO to use an optimistic pipeline for high throughput. *ParaLyze* optimistically precomputes read/write sets, which are used by *ParaFramer* to generate transaction streams with high TLP. *ParaFramer* uses *ParaBloom* to detect conflicts in a cache-friendly CPU-efficient manner, and amortizes conflict checks across multiple transactions using *frames*. *ParaScheduler* uses frames to efficiently compute and execute the optimal conflict-serializable schedule of the transaction stream, identifying opportunities for concurrent execution of transactions within and across frames.

3 Evaluation

We demonstrate FAFO's high throughput and scalability even on workloads with large state sizes and high contention.

3.1 Setup

Hardware All benchmarks run on a single AWS `i8g.metal-24xl` instance with 96 vCPUs (ARM-based Graviton3 cores), 768 GiB of DRAM, and 6 local NVMe SSDs in a RAID0 configuration (aggregated 22.5 TB, 2.16 M IOPS, 26 GB/s read and 20 GB/s write).

3.2 Workloads

We evaluate FAFO using two synthetic workloads: *Native* transfer and *ERC20* transfer. *Native* transfers ETH from one account to another, changing two state slots (the sender balance and receiver balance). *ERC20* transfers an ERC20 token from one holder to another, changing three state slots (the sender ETH balance, the sender token balance, and the receiver token balance). We only run *ParaLyze* on a subset of the ERC20 transfers, to simulate the behavior of the system if EIP-2930 [4] is implemented.

We generate batches of transfers parameterized by *skew* and *contention ratio*, described below.

- γ **(Skew):** The number of *hot* addresses that receive transfers more frequently. Lower γ exacerbates storage hotspots and reduces maximum TLP.

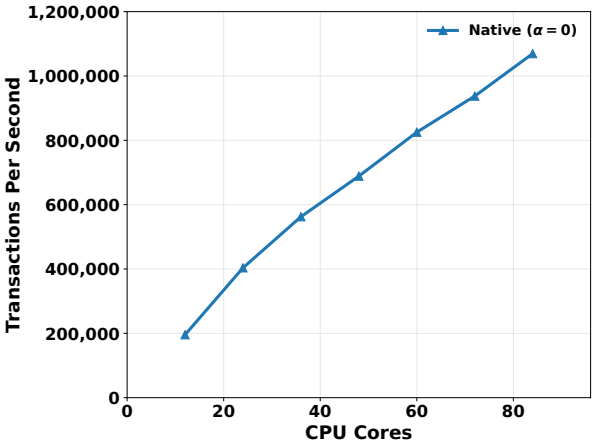

Figure 3: **FAFO scales linearly.** FAFO extracts about 130 TLP from $native_{\alpha=0}$, theoretically allowing it to scale up to 130 CPU cores. FAFO efficiently uses each additional CPU core, up to the maximum available (96).

- α **(Contention Ratio):** The probability of transferring to a hot address.

We represent the specific benchmark parameterization of workload (e.g., ERC20), skew Y, and contention ratio A as $ERC20_{\alpha=A,\gamma=Y}$ (if $\alpha = 0$, we omit γ for brevity).

The `from` address is uniformly randomly chosen from the set of *hot* addresses with probability α, and chosen from the remainder of the addresses with probability $1-\alpha$. The `to` address is always uniformly randomly chosen from the non-hot addresses. Hot keys are chosen (uniformly) from a narrow, contiguous range of size γ from the beginning of the lexicographically sorted list of addresses.

In all benchmarks, we pre-populate the state with 2^{30} (approx. 1 billion) accounts, then issue 500 K batched transfers on each of 512 threads (256 M transactions total). We then issue 512 additional *transfer blocks*, each with approximately 500,000 batched transfers. For both *native* and *ERC20*, each transfer is represented as a 24-byte record (6 bytes sender address, 6 bytes receiver address, 12 bytes of offsets), with the address of the global ERC20 contract being implicit.

3.3 Results

FAFO scales with additional CPU cores Fig 3 shows that FAFO throughput scales **linearly** with each additional CPU core available; this trend is expected to plateau when the number of cores exceeds the TLP of the underlying workload.

FAFO is 91% cheaper than sharding. FAFO achieves over 1 million TPS at less than 10% of the

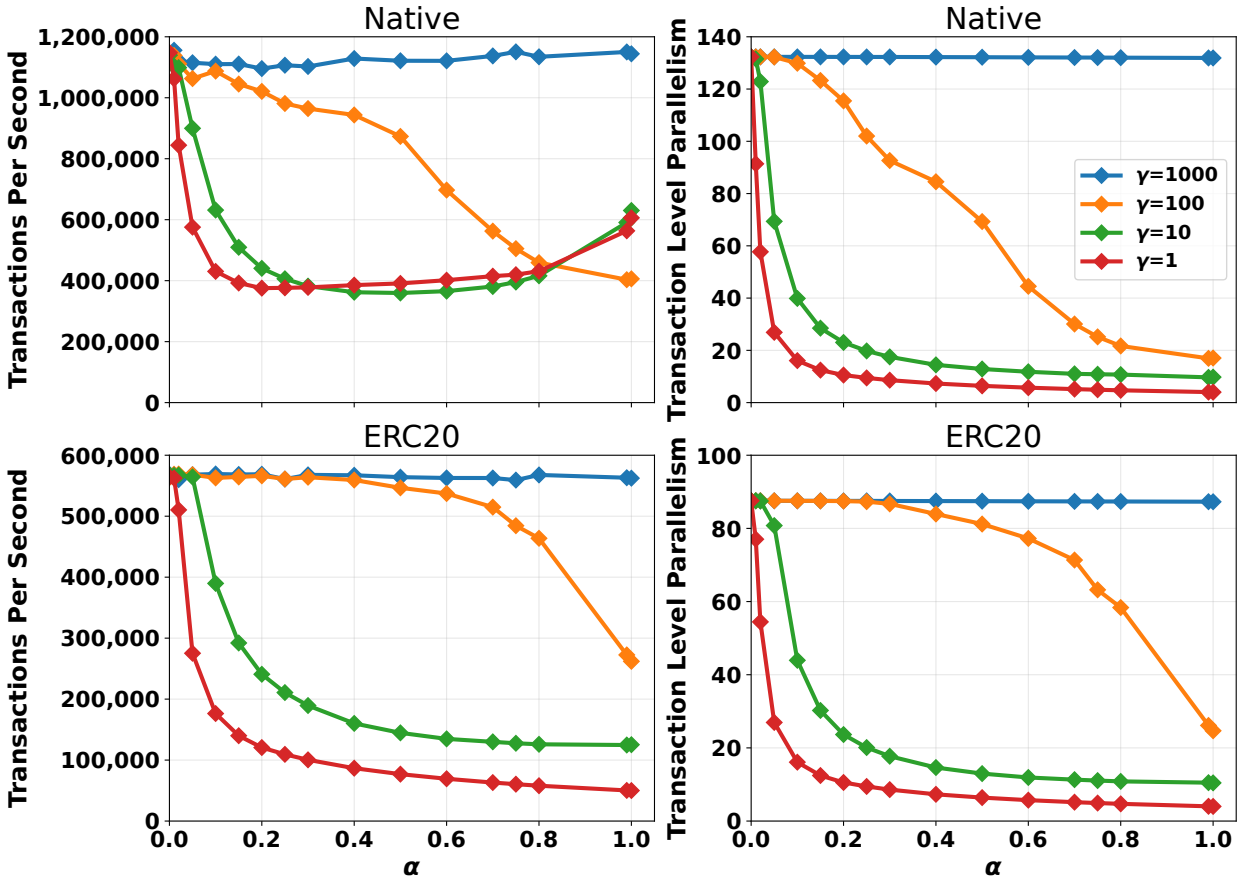

Figure 4: We run FAFO under varying combinations of α and γ (§ 3.2) on a total dataset size of 2^{30} keys. Higher contention reduces TLP and TPS, but FAFO achieves over 1.1 million TPS even with 99% of requests hitting only 1000 keys (approx one per million).

cost of the state-of-the-art sharding-based approach (approximately 6,013 vs 65,361 USD per month based on monthly on-demand pricing for AWS [1] and GCP [10] at the time of writing).

FAFO handles contention well We run the *transfer* workload with varying values of α and γ, measuring the TLP and TPS of FAFO (Figure 4. Even under unrealistically adversarial conditions (99% of requests accessing 0.0001% of the storage slots), FAFO maintains over 130 TLP and 1.1 million TPS. We believe that real-world workloads will be much less skewed, with real blockchain hotspots only being 0.1% of storage slots accounting for 62% of accesses (Section 4). This 0.1% of storage slots corresponds to $\gamma > 1,000,000$, which is omitted from our results as it is indistinguishable from $\gamma = 1000$. The high throughput of FAFO coupled with the massive state sizes enabled by QMDB makes it unrealistic for any real workload to have fewer than 10,000 or even 100,000 hot storage slots.

The unusually high TPS of FAFO for small values of γ and large values of α is likely due to extremely high cache locality of the execution client workload when the context for all hot accounts fit into L1 cache.

For the *ERC20* workload, the throughput of FAFO drops end-to-end by about 50%; this is because each ERC20 transfer writes 50% more state (3 entries vs 2 entries) and requires more computation per transaction (contract vs native).

FAFO has low CPU overhead We perform an ablation study to compare the CPU overhead of FAFO. Our benchmark with only *ParaFramer* and *ParaScheduler* (skipping transaction execution) schedules over 2 million TPS.

4 Related Work

Increasing execution layer parallelism A challenge facing modern blockchains is the *hotspot problem*, where

0.1% of storage slots account for 62% of accesses [12], resulting in high contention that limits parallelism. This has limited the efficacy of existing approaches that rely on optimistic concurrency control (OCC) to modest speedups of just 2–5× on EVM workloads [16, 12]. Sharding approaches [11] have seen some success scaling horizontally, but introduce new synchronization and storage bottlenecks (exacerbated by the hot spot problem).

Shardines [11] scales Block-STM but runs into storage bottlenecks at 30 nodes and observes sublinear scaling and diminishing marginal returns per additional shard with a 33% drop in efficiency for workloads with contention when the number of shards was doubled. In our work, we demonstrate that it is possible to achieve more than enough throughput to serve current blockchain applications without introducing the complexity or cost of sharding. However, sharding is an orthogonal approach to FAFO, and these approaches can be composed if necessary.

Parallel Execution in EVM Blockchains. Prior attempts at accelerating Ethereum's sequential execution rely on optimistic concurrency control (OCC) and speculative execution. Block-STM [9] applies multi-version OCC to speculatively run transactions in parallel, which boosts throughput but has been shown to suffer from high abort overhead under high contention [12]. ParallelEVM [12] opts for more fine-grained operation-level concurrency and uses a static single-assignment (SSA) log to trace operation dependencies, identify conflicts and re-execute, with a reported speedup of 4.28×. Forerunner [6] speculatively executes transactions in the window of time between dissemination and executes multiple possible futures in parallel execution, using memoization to speed it up.

Transaction reordering Replica-side schedulers such as OptME [15] and DMVCC [13] reorder *statements* after the block is disseminated. Because every replica must reproduce the identical schedule, they either (i) speculatively execute each transaction to discover its read/write set (high overhead) or (ii) require users to supply access lists (poor UX). They also cannot embed proposer-side policies (anti-spam, local fee markets), and any algorithmic change requires a hard fork. In contrast, FAFO block producers reorder *transactions* before block formation, reducing per-validator CPU overhead and enabling arbitrary scheduler upgrades. Hyperledger-style systems (e.g., Fabric++ [17], FabricSharp [14], and HTFabric [18]) reorder to minimize aborts inside an Execute-Order-Validate model, but this EOV model is orthogonal to the Order–Execute architecture of account-based blockchains.

Balancing static analysis and speculative execution Solana [21] avoids aborts entirely through lock-based scheduling on static read/write sets, but offloads the burden of accurate resource specification to contract developers. At the other extreme, Dickerson *et al.* [7] and OptSmart [2] propose speculative scheduling on the block producer; this lowers validator conflicts in exchange for increased CPU overhead on the leader. FAFO's hybrid approach strikes a better balance between these two extremes.

5 Conclusion

We introduce FAFO, a transaction scheduler that schedules and executes over 1.1 million EVM transactions per second on a single node . FAFO delivers near-linear throughput scaling with CPU core count and achieves the same throughput as sharded systems with 91% lower cost.

FAFO departs from speculation-reliant models by reordering transactions *before* block formation to better exploit opportunities for parallelism. This maximizes concurrency and minimizes synchronization overhead by combining efficient approximate conflict detection (*ParaBloom*) with lightweight static scheduling (*ParaScheduler*).

By reducing the cost of high-throughput execution and lowering capital costs, FAFO enables high throughput without sacrificing decentralization.

References

[1] Amazon Web Services. Amazon ec2 on-demand pricing. Accessed: 2025-06-23.

[2] Anjana, P. S., Kumari, S., Peri, S., Rathor, S., and Somani, A. Optsmart: a space efficient opt imistic concurrent execution of smart contracts. *Distributed and Parallel Databases 42*, 2 (2024), 245–297.

[3] Block.one. EOS.IO Technical White Paper v2, March 2018. Accessed: 2025-02-27.

[4] Buterin, Vitalik and Martin Swende. Eip-2930: Optional access lists, 2020. Ethereum Improvement Proposal.

[5] Cellary, W., Morzy, T., and Gelenbe, E. *Concurrency control in distributed database systems*, vol. 3. Elsevier, 2014.

[6] Chen, Y., Guo, Z., Li, R., Chen, S., Zhou, L., Zhou, Y., and Zhang, X. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 570–587.

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

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

[9] GELASHVILI, R., SPIEGELMAN, A., XIANG, Z., DANEZIS, G., LI, Z., MALKHI, D., XIA, Y., AND ZHOU, R. 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.

[10] GOOGLE CLOUD. Compute engine pricing. Accessed: 2025-06-23.

[11] LABS, A. Shardines: Aptos' sharded execution engine blazes to 1m tps, 2025. Archived on 2025-02-11.

[12] LIN, H., FENG, H., ZHOU, Y., AND WU, L. Parallelevm: Operation-level concurrent transaction execution for evm-compatible blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems* (2025), pp. 211–225.

[13] QI, X., JIAO, J., AND LI, Y. Smart contract parallel execution with fine-grained state accesses. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)* (2023), IEEE, pp. 841–852.

[14] RUAN, P., LOGHIN, D., TA, Q.-T., ZHANG, M., CHEN, G., AND OOI, B. C. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 543–557.

[15] RYU, D., AND PARK, C. Toward high-performance blockchain system by blurring the line between ordering and execution. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis* (2024), IEEE, pp. 1–16.

[16] SHAHID, R. Parallel transaction execution in public blockchain systems. Master's thesis, University of Waterloo, 2024.

[17] SHARMA, A., SCHUHKNECHT, F. M., AGRAWAL, D., AND DITTRICH, J. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 105–122.

[18] SONG, J., JEONG, J., LEE, J., NA, I., AND KIM, M.-S. Ht-fabric: A fast re-ordering and parallel re-execution method for a high-throughput blockchain. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management* (2024), pp. 2118–2127.

[19] THE MYSTENLABS TEAM. The sui smart contracts platform.

[20] WOOD, G. Ethereum: A secure decentralized generalized transaction ledger. In *Ethereum Yellow Paper* (2014).

[21] YAKOVENKO, A. Solana: A new architecture for a high performance blockchain v0. 8.13, 2018.

[22] ZHANG, I., ZARICK, R., PELLEGRINO, B., LI, T., WONG, D., KIM, T., ROY, U., GUIBAS, J., AND KULKARNI, K. vapps: Verifiable applications at internet scale. *arXiv preprint arXiv:2504.14809* (2025).

[23] ZHANG, I., ZARICK, R., WONG, D., KIM, T., PELLEGRINO, B., LI, M., AND WONG, K. Qmdb: Quick merkle database. *arXiv preprint arXiv:2501.05262* (2025).