

REMORA: Scale-out Deterministic Execution for Smart Contracts

Zhengqing Liu¹, Alberto Sonnino^{2,3}, Igor Zablotchi², Eleftherios Kokoris Kogias², Marios Kogias¹

¹Imperial College London, ²Mysten Labs, ³University College London

ABSTRACT

Modern blockchains rely on a modular architecture that decouples consensus from execution. Recent advances in consensus algorithms have shifted the bottleneck to the execution layer, which must deterministically follow the consensus order and handle increasingly complex, compute-intensive smart contracts. We identify that single-node validators cannot keep up, motivating the need for a scale-out design.

We design REMORA, a scale-out smart contract execution engine. REMORA adopts an efficient asymmetric architecture with centralized transaction dispatching and distributed execution, and depends on an object versioning scheme with a strict ownership model to guarantee deterministic scale-out execution. REMORA achieves up to 3× throughput improvement compared to state-of-the-art deterministic execution schemes, scales up to 250k TPS, matching modern consensus performance, and reduces latency by up to 5ms. We also show that REMORA elastically adapts to bursty workloads and dynamic access patterns using real-world traces. REMORA’s main performance benefits come from a novel stateless-stateful separation during smart contract execution, which overlaps the execution of state-independent tasks with consensus, and a new locality-aware and load-balanced scheduling scheme.

1 INTRODUCTION

Modern blockchains are operated by *validators*, which adopt a layered architecture (Figure 1): the *consensus* layer establishes a globally ordered sequence of transactions, and the *execution* layer performs transaction execution. While enormous progress has been made in scaling consensus, with production systems today sustaining 200k–300k transactions per second (TPS) [7, 8], the execution layer has failed to keep pace. The need for faster execution is amplified by the rise of smart contracts, which drive new demands from decentralized applications spanning finance, gaming, and identity [37, 38, 79, 83]. Thus, execution, not consensus, now defines the scalability frontier for blockchains.

Despite advances in exploiting multi-core parallelism on smart contract VMs [36, 67, 86, 102], *scale-up* execution alone cannot meet the computational demands of modern smart contracts. As smart contract logic and cryptographic authentication grow more expensive, due to techniques like zero-knowledge proofs and post-quantum cryptography [3, 57, 98], scaling-up becomes impractical as computational needs exceed the capacity of single-node validators. Addressing this bottleneck requires a shift to *scale-out* designs that distribute validator’s execution across multiple machines.

A core requirement of blockchain execution that makes the problem more challenging compared to prior work on distributed transaction processing [48, 62, 116], is *strict determinism* [20], i.e., preserving the total order established by the consensus layer during execution even in the presence of parallel or distributed executors. In blockchains, replicas are mutually untrusted, and transaction order often carries financial significance in applications like auctions

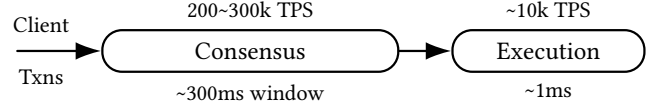


Figure 1: Validator architecture in modern blockchains.

and flash loans. Enforcing the consensus-established order is essential because it guarantees fairness, preserves transparency, and ensures that execution remains verifiable and tamper-proof [70, 89].

In this work, we ask: *how to efficiently scale-out smart contract execution while preserving strict determinism?* Revisiting prior work on deterministic transactional systems, we identify two fundamental design pitfalls. First, all prior deterministic databases adopt a *symmetric* architecture, that duplicates the full control stack, i.e., sequencing and scheduling all transactions, on every node, while only performing execution for a fraction of them [78, 80, 85, 92, 99]. This redundancy inflates the coordination overhead and leads to limited scalability. Second, there is currently no distributed transaction-scheduling scheme that jointly satisfies strict determinism, locality awareness, and load balance, properties that are necessary for efficient scale-out smart contract execution. Load-driven schemes trigger excessive state migration and coordination for distributed transactions, while locality-driven ones concentrate records on a single node [50, 68]. Recent work attempts to balance both [69, 92], but often breaks strict determinism via reordering, which is unacceptable to blockchain systems.

Beyond identifying design pitfalls in prior work, we further observe two new domain-specific opportunities by leveraging blockchain characteristics. First, smart contract execution naturally comprises two steps: a *stateless* one, performing verification and authentication, which is compute-intensive but independent of shared state; and a *stateful* one, performing business logic of smart contracts, which thus requires access to the blockchain state but tends to be lightweight. Accessing shared state introduces dependencies and ordering constraints across transactions. However, we argue that the stateless parts have no such dependencies and can be executed independently on any available compute resources, outside the critical dependency boundaries. Second, the stability of existing consensus algorithms introduces a *window* of opportunity between transaction proposal and commitment with high predictability in proposal outcomes, which allows for speculative off-path execution to perform useful work ahead of time [18]. We pinpoint the parts of the validator execution that can be efficiently performed during the consensus window, without incurring excessive state transfers and cascading aborts in a scale-out setting.

Driven by these findings, we design REMORA, a scale-out execution engine for blockchain systems that preserves strict determinism. REMORA adopts an *asymmetric* architecture, with a single Coordinator managing a pool of Workers to decouple scheduling from execution. The Coordinator receives the globally ordered transaction stream from consensus and dispatches each transaction to a Worker for execution. To enforce strict determinism, REMORA

introduces *per-object versioning* and a *lease-based ownership* model: each versioned object is exclusively owned by either the Coordinator or a single Worker at any point in time. Each transaction is annotated with an explicit read/write set, and the Coordinator assigns object versions in consensus order. When a transaction requires objects owned by a different Worker, REMORA transfers leases and object state accordingly. This design simplifies the enforcement of determinism and, by allowing dynamic flow of object ownership, enables scheduling optimizations that colocate related transactions and improve locality.

REMORA decouples the execution of stateless and stateful parts, proposing *pre-consensus stateless execution* and *stateful scheduling* to gain both latency and throughput benefits. The separation allows compute-heavy stateless parts to harness available resources within the cluster. To handle stateful parts under contended and skewed workloads, REMORA introduces a *subgraph-first scheduling (SFS)* scheme that focuses on scheduling disconnected subgraphs within the transaction dependency graph, jointly considering transaction ordering, load distribution, and data locality. To tolerate worker failures, REMORA further employs periodic snapshots to persist batched updates. REMORA also implements a Coordinator-driven elasticity scheme that adds or removes nodes to the cluster and depends on lazy state transfers to the new nodes.

We evaluate REMORA using transactional benchmarks and real-world smart-contract traces, showing up to $3\times$ throughput improvement compared to state-of-the-art deterministic execution scheme [69], while effectively masking the latency of both stateless execution and stateful scheduling in consensus window, thus reducing end-to-end latency by up to 5ms. REMORA’s asymmetric architecture also demonstrates higher efficiency than traditional deterministic databases [78, 99] as system scales, thus reducing deployment cost. Our implementation sustains over 250k TPS even in the presence of split execution and the computationally heavy SFS scheduling, matching the modern consensus throughput.

This paper makes the following contributions:

- Two blockchain-specific design opportunities (§3), i.e., decoupling stateless and stateful execution, and enabling pre-consensus stateless execution and stateful scheduling to improve performance in a scale-out setting (§4.3, §4.4).
- An object-versioning and lease-based ownership model to preserve strict determinism in a distributed setting and simplify deterministic parallel execution in each Worker (§4.2).
- SFS, a subgraph-first scheduling scheme that jointly considers load, locality, and dependency ordering for efficient execution of smart contract workloads (§4.5).
- REMORA, a scale-out execution engine for smart contracts that adopts an asymmetric architecture, improving efficiency over prior deterministic distributed systems and addressing blockchain-specific requirements (§4.1).

2 BACKGROUND AND MOTIVATION

In this section, we describe the high-level architecture of modern blockchains, identify the scalability bottleneck, and motivate the need for scale-out. We then outline the unique requirements of blockchain infrastructure and explain why existing (non-) deterministic distributed transaction systems fall short of meeting them.

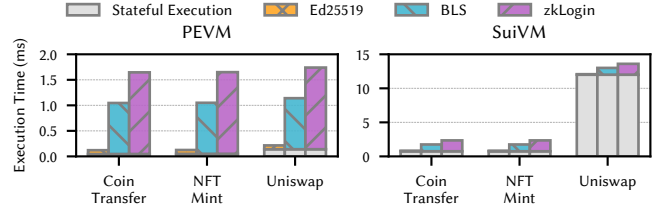


Figure 2: Execution breakdown of common smart contracts.

2.1 Blockchain Architecture and Bottlenecks

Modern blockchains typically adopt modular architectures that decouple consensus from execution [55, 59, 60]. Validators first organize transactions in blocks and propose them to the network. These blocks are then ordered by a consensus protocol before being executed deterministically by the validators’ execution engine. This decoupling enables independent evolution of the two layers and allows diverse combinations of technologies. For example, both Aptos [54] and Cronos [56] adopt BlockSTM [36] for execution, despite relying on different consensus protocols.

This architectural decoupling also means that either component, consensus or execution, can independently become a bottleneck. Although early consensus protocols only scaled to a few hundreds or thousands of TPS [14], modern designs now sustain over 300k TPS by allowing validators to propose transactions in parallel and leveraging state-of-the-art BFT protocols [7, 8]. As a result, *the bottleneck has now shifted to execution*. Despite extensive parallelization efforts [36, 39, 40, 58, 71], end-to-end throughput in practice remains at tens of kTPS due to the smart contract complexity and the overhead of the underlying VMs [11].

To better understand the execution bottleneck, Figure 2 shows the average execution time of several smart contracts on two virtual machines [59, 86]. Depending on the contract logic and the verification method, the execution can take up to several milliseconds of compute time. Even under ideal assumptions with a modest execution time of 1 ms per transaction, uniform load with limited inter-transaction dependencies, Little’s Law implies that sustaining 300k TPS (the throughput of state-of-the-art consensus protocols) would require roughly 300 cores dedicated to execution. This far exceeds the capacity of typical validator hardware [94–96].

A broad line of work tackles such bottlenecks via *blockchain sharding*, where the blockchain state (and hence execution) is partitioned across shards maintained by different validator committees in parallel [5, 25, 41, 44, 52, 75, 115]. However, sharding introduces recurring challenges. First, it relies on subcommittee formation through random sampling which needs stronger adversarial assumptions to keep every shard secure with high probability [25, 52, 115]. Second, transactions that span shards incur substantial overhead due to Byzantine-resilient cross-shard coordination (often akin to atomic commit), which can dominate performance under realistic cross-shard access patterns [113], since state migration requires subcommittee-to-subcommittee BFT coordination. These challenges make sharding operationally complex, and recent production roadmaps have decided to shift away from sharding [28].

We argue there is an alternative scale-out approach: *scale out execution within a single validator*. Rather than repartitioning state

across multiple validator committees, we can focus on the execution bottleneck of a single validator and distribute the workload across a cluster of machines, while preserving the consensus protocol, validator set, and trust assumptions. This design allows a validator, which is a single administrative and trust domain, to scale its execution capacity elastically with available compute resources, narrowing the widening gap between consensus throughput and single-node execution.

Takeaway 1: *The computational need of modern smart contracts goes beyond the capacity of single-node validators, making the scale-out design necessary. Scale-out intra-validator execution can retain the trust model and operational requirements of modern blockchains, unlike inter-validator sharding approaches.*

2.2 Design Requirements and Prior Work Limitations

Focusing on a scale-out architecture of a *single validator*, we enumerate the key requirements that guide such a design. To do so, we revisit related work across blockchains and distributed databases/systems to highlight where prior approaches fall short and to surface the practical considerations for real deployments.

R1: Strict Determinism. Preserving the consensus-established order, i.e., strict determinism [20], is the fundamental correctness requirement for blockchain execution. Given a globally agreed-upon transaction order, a strictly deterministic execution layer guarantees that the resulting effects are identical to those of sequential execution. This corresponds to *strictly-deterministic serializability* [20]: for an order of transactions $O = \langle T_1, \dots, T_n \rangle$, the execution effect is equivalent to sequentially executing and committing O in that order. Other prior work also formalizes the specific total order into the correctness target known as *Byzantine Ordered Consensus* [120, 121]. This requirement sharply differentiates blockchains from conventional deterministic systems within a trusted administrative domain, where controlled reordering can be used to improve performance by reducing dependencies or object transfers [69, 73, 92].

In a blockchain setting, strict adherence to the consensus order is critical for two reasons. First, replicas are operated by mutually untrusted parties, and the established order of transactions can have direct financial consequences in order-sensitive applications (e.g., auctions, flash minting) [70, 89]. Second, blockchains are designed as permanent, decades-spanning ledgers requiring continuous public auditability. If execution engines were allowed to apply different reorderings over time, validators would need to record additional per-block metadata to reproduce the chosen order, complicating verification and weakening the replayability. Strict adherence to consensus order is thus the current industry standard.

R2: Efficiency and Cost-Effectiveness. A scale-out execution layer should convert added machines into execution throughput, not duplicated control-plane work. Calvin [99] pioneered distributed deterministic execution by enforcing a global transaction order to eliminate two-phase commit. In Calvin, every node receives the fully replicated, ordered batch of transactions, even though each node manages only a partition of the object space. To maintain determinism, nodes process only the transactions relevant to their partition, strictly following the global order. Transactions execute

in-place on partitioned state, with remote reads and coordination handled during batched phases.

Calvin-style designs have been widely followed by later systems [19, 69, 78, 80, 85, 92], which keep a *symmetric* architecture: every node sequences, schedules, and executes. This preserves determinism but duplicates work, i.e., each node processes the full batch of transactions, runs the same dependency analysis, and schedules, even though it executes only a subset. The result is wasted CPU, higher coordination overhead, and higher per-node provisioning cost (full metadata/control stack on every machine). As the cluster or throughput grows, these replicated costs dominate, so scale-out shows diminishing returns (quantified in §6.2).

R3: Workload Adaptiveness. A scale-out execution layer should adapt online to workload skew and hotspots, minimizing distributed transactions by favoring locality, while still balancing loads. Prior work shows that performance in distributed databases is highly sensitive to data partitioning [69, 92], as it determines the fraction of distributed versus single-node transactions. Although offline profiling or periodic repartitioning can improve locality [88, 92], real workloads are dynamic and hard to predict [81, 104, 118]. Even workload-driven repartitioning and live object migration [2, 24] can fall short in complex deployments, as prior work [69] highlights. Hence, recent systems combine repartitioning with on-demand migration so that objects flow across nodes according to the workload needs as part of transaction processing [50, 68, 69].

However, freely allowing object migrations across nodes can lead to further inefficiencies. On one hand, load-driven policies, which try to equalize the load among nodes, trigger frequent state transfers and “ping-pong” effects [45, 69], causing stalls and inefficiency. On the other hand, locality-driven policies, which seek to minimize object transfers, collapse active records onto a single node in the presence of skewed workloads, as shown in §6.3. Skewed workloads are the norm in modern blockchains [10, 67]. Some designs consider both locality and load, but are either non-deterministic [122] or heavily rely on reordering [69, 92], thus preventing their use in blockchain systems that require strict determinism.

R4: Fault Tolerance. Scale-out execution introduces additional failure modes: execution nodes may crash or be temporarily unavailable. Our goal is to scale *intra-validator* execution, without changing the consensus layer. BFT remains provided by the blockchain’s consensus across validators, and our design should not change its fault threshold or safety guarantees. Given that one validator is a single administrative and trust domain, we target *crash fault tolerance* for execution nodes. Specifically, we aim to preserve availability and avoid reducing a validator’s mean-time-to-failure (MTTF) due to the increased number of components that can fail independently, rather than to defend against intra-validator compromise.

R5: Elastic Autoscaling. Production workloads are bursty and time-varying. Thus, a practical execution layer should be elastic to scale resources up/down and adapt placement online without disruptive global reconfiguration. Past permissioned systems [104–106] emphasize the need for elasticity, yet they focus on the *whole chain* and make substantial and potentially non-compatible changes to the execution, architecture, and all validators. This work instead targets *single-validator* elasticity arguing that each validator should

scale its execution capacity independently without changing the consensus layer or affecting other validators.

Takeaway 2: *Scale-out blockchain execution requires a careful co-design for strict determinism, architecture efficiency, workload adaptiveness, fault tolerance, and elasticity as independent design choices might lead to requirement violations.*

3 REMORA INSIGHTS

Before diving into the REMORA design, we highlight two crucial observations that are unique to blockchain systems and substantially influence the proposed architecture.

3.1 Stateless-Stateful Separation

We observe that smart contract execution naturally decomposes into two components: *stateless* operations, dominated by cryptographic verification of transaction authenticity, and *stateful* operations, which execute the contract logic and modify the blockchain state. This separation creates an opportunity to scale stateless work independently of stateful execution. Stateless work is completely independent of the smart contract logic or the blockchain state, imposing no ordering or placement constraints. Hence, it can be executed on any available compute resource in the validator cluster.

Figure 2 breaks down the execution time of three representative smart contracts: a simple transfer, an NFT mint, and a Uniswap [72] trade. Two smart contract VMs are used: SuiVM [59], a production-grade MoveVM powering the Sui [58] blockchain, and PEVM [86], a recent parallel EVM implementation written in Rust. We pair each stateful part with one of three primary authentication methods [57] in modern blockchains: (1) simple signatures (e.g., Ed25519 [13] or BLS [12]), (2) key-less authentication via external identity providers (e.g., zkLogin [9]), and (3) multi-signatures combining several instances of these methods. Differences in stateful execution time across VMs reflect their varying runtime overheads and optimizations—lighter VMs reduce execution costs by design, making the stateful component more efficient.

We observe that stateless compute often dominates total execution time of the smart contract: 1 ms for BLS [15, 66], 1.6 ms for zkLogin [15, 34], and up to 16 ms for multi-signatures involving ten zkLogin proofs. While simple signatures remain most common, key-less authentication is rapidly gaining adoption. We observe roughly 10k zkLogin-based transactions per day on a production blockchain validator [58]. Multi-signatures are also used, albeit less frequently, with about three transactions per minute requiring verification of multi-signatures comprising two to ten signatures or zkLogin proofs. As on-chain workloads expand to include post-quantum cryptography, the stateless portion of execution is expected to become increasingly compute-intensive [3, 38, 98].

Insight 1: Smart contract execution consists of a stateless and a stateful part, with the stateless portion being increasingly compute-intensive yet runnable without ordering or placement constraints on any available compute resource.

3.2 Consensus Window

Prior work [18] has identified there is a window between when a transaction is proposed and when it is finally committed that can be leveraged for speculative execution to improve performance of the blockchain. In state-of-the-art production-grade consensus systems [8], this window can last over 300ms, which is a substantial amount of time compared with smart contract execution time.

To evaluate the opportunity, we study existing blockchain consensus algorithms and deployed systems and observe the following. Existing consensus protocols exhibit a high degree of predictability in practice: proposals are committed in the expected order in the overwhelming majority of cases. Deviations occur only when (i) a block proposal is invalid (e.g., due to faulty validators), or (ii) network asynchrony delays or reorders commitments. Empirical evidence from Sui’s L1 blockchain infrastructure shows that 98.66% of blocks are committed directly in the proposed order [58]. Moreover, after a block proposal collects a quorum of votes [14] in the first phase of the protocol [8, 14, 91, 112], only extreme network asynchrony can prevent it from being committed as expected.

Leveraging the consensus window becomes considerably more challenging in the context of scale-out execution. Prior work [18, 117] exploits this window for speculative smart contract execution. This is feasible in their setting because they assume a single-node validator, which holds the entire blockchain state locally. In a scale-out setting, however, state is partitioned across nodes. Speculative execution would then require speculative object transfers across partitions to resolve dependencies. These transfers not only impose significant communication cost, but also create the risk of cascading aborts: a single misprediction in the final commit order can invalidate multiple speculative transfers and their dependent computation across nodes. The combined cost of speculative state movement and large-scale rollbacks can easily outweigh the performance gains, rendering naive scale-out execution impractical.

To still harvest the benefits of the consensus window, we identify two parts of validator execution that can be performed speculatively without requiring access to the underlying blockchain state. These are: (i) the stateless part of a smart contract and (ii) the scheduling logic that determines which transactions each node will execute. The first requires only the transaction input, while the second relies on placement metadata and prior scheduling decisions. Crucially, neither depends on the distributed blockchain state. By only optimistically performing these state-independent components, we capture the benefits of the consensus window without costly transfers or cascading aborts.

Insight 2: In a scale-out design, validator tasks that can be efficiently performed in the consensus window are stateless execution and transaction scheduling.

4 THE REMORA DESIGN

We design REMORA, a scale-out execution engine for a single validator, to cater to the needs of modern blockchain infrastructure. REMORA should: (i) follow a modular architecture that decouples consensus from smart contract execution and preserve strict determinism; (ii) leverage stateful and stateless execution separation; (iii) allow to implement various pre- and post-consensus scheduling

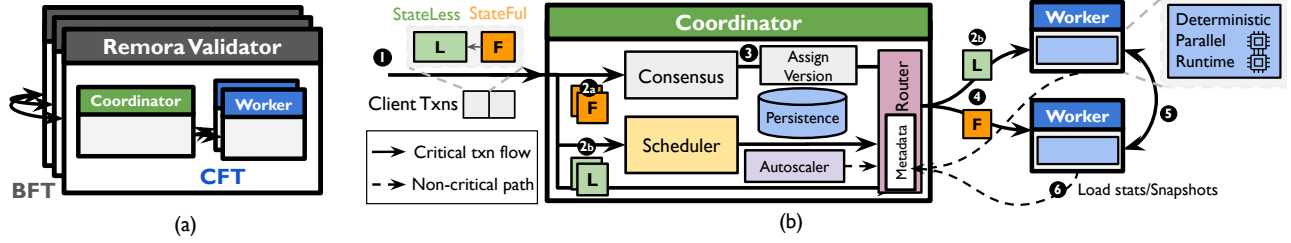


Figure 3: REMORA Architecture. REMORA is a single validator inheriting Byzantine Fault Tolerance (BFT) from the consensus layer (a), and internally scales out execution (b) with a centralized Coordinator and a pool of Workers under Crash Fault Tolerance (CFT).

algorithms; (iv) enable seamless elasticity by dynamically adapting to load without upfront costs; (v) tolerate execution node failures within a trusted validator. We provide formalized proofs in the Appendix.

4.1 High-Level Overview

Architecture. REMORA adopts an *asymmetric* architecture that splits validator functionality across specialized nodes, in contrast to prior symmetric designs (§2.2) where every node redundantly performs ordering, scheduling, and execution. A single Coordinator node handles sequencing and scheduling, while a pool of Worker nodes split the object state among them and execute smart contracts. The Coordinator is the only participant in the consensus protocol, from which it receives the ordered sequence of transactions. Then, based on the scheduling logic, it dispatches transactions to the Workers for deterministic scale-out execution. The Coordinator also manages elasticity and resource allocation, durably persisting execution results, and coordinating failure recovery.

Workflows. Figure 3 gives an overview of the main components and the transaction execution workflow in REMORA. The Coordinator receives transactions from clients (1), and the consensus module runs a sequencing protocol to determine their global order according to the specific blockchain deployment. REMORA separates the stateless and stateful components early: it forwards the stateful part to the scheduler for speculative planning (2a), and dispatches the stateless part directly to Workers according to the scheduling policy (2b). Both steps leverage the consensus window to overlap with consensus and reduce latency. After consensus finalizes the transaction sequence, REMORA assigns object versions to each transaction (3) to enforce a deterministic execution schedule preserving the established order. The Coordinator then routes the stateful components to Workers using the scheduler’s precomputed plan. Workers execute only the transactions that the Coordinator assigns to them (4) and communicate with their peers when remote state is needed (5). They also periodically report load and execution state snapshots (6), enabling autoscaling to adjust cluster size based on utilization. The persistence module in the Coordinator safely stores these snapshots to ensure crash recovery for Workers, alongside the transaction logs determined by the consensus.

This asymmetric design effectively fulfills our design goals. First, the consensus layer is agnostic to REMORA, making the architecture reusable across different blockchain deployments. Second, concentrating scheduling in the Coordinator avoids duplicating scheduling logic at every Worker and enables new policies without Worker

changes. Third, with its global view of cluster utilization, the Coordinator can make accurate autoscaling decisions. Finally, persisting results asynchronously combined with deterministic execution allows for efficient Worker fault tolerance.

Threat Model. REMORA scales out execution within a single trust domain, the validator. A REMORA deployment should therefore be viewed as a *single consensus participant* implemented by a Coordinator and a pool of internal Worker machines. The Coordinator is the validator’s sole externally visible interface to consensus, and any fault suffered by the Coordinator is applied as a total domain-wide failure as far as the blockchain protocol is concerned. Workers, in contrast, are internal execution resources under the same administrative trust domain as the Coordinator (analogous to threads/cores in a scale-up validator), so we assume they are trusted but crash-stop, i.e., they may fail by crashing, but do not behave maliciously. Worker failures do not affect the blockchain protocol’s security as they only affect the validator’s performance.

4.2 Enforcing Strict Determinism

REMORA assumes a widely used smart contract model [71, 77] in which contracts execute atop a flat key-value store and provides ACID transactional guarantees. Each contract declares its read and write sets upfront, an assumption shown to hold for most transactional workloads (further discussed in §7). The Coordinator acts as the global owner of all objects but delegates execution by leasing objects to Workers. Each Worker holds the leased state in memory and periodically reports updates back to the Coordinator. The Coordinator maintains metadata tracking the current leaseholder for each object. Based on the scheduling policy, the Coordinator forwards a transaction to a single Worker for execution. If all objects reside in the delegated Worker, execution runs to completion. Otherwise, the Worker initiates lease transfers and fetches remote objects from their current owners before execution.

Object Versioning. REMORA’s scale-out design introduces challenges in guaranteeing deterministic execution in a distributed setting, beyond the scope of single-node deterministic parallel execution [33, 36, 71]. To enforce strict determinism in a scale-out manner, REMORA leverages the Coordinator as a serialization point. For each transaction in the consensus-ordered batch, the Coordinator assigns a unique version to every accessed object (read or write) by incrementing its per-object counter. This ensures that all accesses to the same object are consistent with the consensus-established order, guaranteeing strict determinism. Version assignment requires only a single linear scan over per-transaction objects within the consensus-ordered batch, incurring negligible overhead. Note that

this version is validator-local and does not have to be the same across different validators. This version-based design is inspired by single-node deterministic parallel systems, such as Bohm [32], where a single thread assigns object versions before execution. REMORA generalizes this principle to distributed execution.

Centralized version assignment simplifies the design of each Worker’s local runtime, which must execute transactions in parallel while preserving determinism. When the Coordinator dispatches a transaction to a Worker, it explicitly defines the specific object versions the transaction needs to access along with their owner node. A transaction does not compute dependencies by identifying the last writer of an object; instead, its dependencies are fully determined by the assigned versions. A transaction becomes runnable once all required versions are available, either locally produced or fetched from remote workers. It observes the shared state defined by input versions, and produces the next versions for its accessed objects. The per-object version stream assigned by the Coordinator ensures that execution respects the consensus order without requiring additional coordination or locking. Furthermore, this naturally enables *inter-block* parallelism [81], as Worker execution proceeds independently of block boundaries.

Object versions have explicit lifetimes, making garbage collection trivial. The Coordinator assigns a unique version per transaction access, ensuring each version is consumed *exactly once*. Once the consuming transaction executes, the Worker safely discards the old version. A new version is created even for reads, which simplifies memory management but limits traditional read-sharing optimizations. The performance cost of this trade-off is minimal: hot objects in blockchain workloads, such as balances, order books, and NFTs, are update-heavy, and prior measurements show that write-write conflicts are far more prevalent than read-write conflicts [10].

Object Ownership and Leasing. REMORA adopts a strict object ownership model: at any point in time, each object resides on a single Worker, which holds an exclusive lease to access and modify it, or the Coordinator. Static partitioning or fixed sharding performs poorly for dynamic workloads with distributed transactions [24], as discussed in §2.2. Thus, instead of periodic repartitioning as in dynamic sharding schemes [2, 88, 114], REMORA transfers ownership on demand through transaction dispatch. This design avoids unnecessary object movement and adapts naturally to access patterns, a strategy adopted by several recent systems [50, 68, 69]. The Coordinator maintains per-object metadata tracking the current version and leaseholder. When scheduling a transaction to a Worker that does not own all required objects, the Coordinator annotates the request with the current owners for each object version. The receiving Worker fetches the necessary state directly from the owning nodes without the Coordinator’s intervention and becomes the new owner. Upon transaction dispatch, the Coordinator updates its metadata accordingly indicating the new Worker as the leaseholder.

4.3 Stateless-Stateful Separation

REMORA leverages a structural property of modern smart contracts: their execution can be naturally decomposed into a *stateless* and a *stateful* part (**Insight 1**). The stateless part performs compute-heavy cryptographic verification and produces a validation token, which serves as an input to the stateful execution. Before executing

the stateful part, the corresponding Worker fetches this token either locally or remotely from the Worker that ran the stateless part. If the stateless validation fails, the transaction is aborted. To maintain version order in such cases, the Worker executing the stateful part performs a no-op that touches the relevant objects, thereby advancing their versions without applying any state changes.

This separation enables different scheduling policies for stateless and stateful parts. The router module dispatches stateless parts with a focus on *load balancing*, since they have no data dependencies. This approach can reactively fill utilization gaps when some Workers are temporarily occupied with stateful execution due to locality-aware scheduling. We describe how REMORA schedules stateful parts in §4.5.

4.4 Leveraging the Consensus Window

Consensus finalization in modern blockchains often takes several hundred milliseconds (e.g., 300 ms). Rather than leaving this interval idle, REMORA exploits it to hide latency and shift expensive work off the post-consensus critical path (**Insight 2**). Specifically, during this window, REMORA performs stateless execution and stateful scheduling, as illustrated in Figure 3 (steps 2a and 2b).

Pre-consensus Stateless Execution. The lack of dependencies in the stateless part allows it to be executed in parallel to consensus. This overlap leverages the consensus window to reduce end-to-end transaction execution latency. While this can lead to wasted computation if a block is eventually rejected, such occurrences are rare, thus will have minimal impact on performance or utilization.

Pre-consensus Stateful Scheduling. In addition to stateless execution, REMORA also runs its stateful scheduling logic during the consensus window. The Coordinator forwards each proposed block simultaneously to the consensus and scheduling modules. While consensus runs, the scheduler analyzes transaction dependencies, and devises a scheduling plan. Overlapping this with consensus allows REMORA to implement even more computationally expensive scheduling algorithms without affecting end-to-end latency and overall throughput. To guarantee correctness, the router dispatches the transactions via pre-computed plan only after consensus finalizes. On the rare occasion that consensus rejects the block, the pre-consensus stateful routing plan is invalidated and dropped. To handle this case, REMORA maintains two versions of metadata: one for the scheduler and one for the router. The router updates its version when dispatching transactions. If a block is dropped, the scheduler pauses, synchronizes the two metadata versions, discards intermediate schedules, and then resumes.

We intentionally avoid pre-consensus stateful execution. Unlike the stateless part, stateful execution is less compute-intensive and tightly coupled with shared state dependencies. Speculating on it would greatly complicate system design, triggering speculative object transfers and cascading aborts in the distributed setting while yielding marginal latency benefits.

4.5 Subgraph-First Scheduling (SFS)

Now we describe how REMORA efficiently schedules stateful transactions among workers. As discussed in §2, an effective scheduling policy must simultaneously satisfy three requirements: strict determinism, locality awareness, and load balance. Existing approaches

Algorithm 1: SFS (Subgraph-First Scheduling)

Input: Batch B (consensus order), per-worker ownership sets $\{O_i\}$, loads $\{P_i\}$
Output: Assignment of subgraphs to workers

- 1 Build dependency graph $D(B)$ and extract subgraphs $\mathcal{G}$
- 2 **foreach** subgraph $G \in \mathcal{G}$ **do**
- 3 $K \leftarrow \bigcup_{t \in G} \text{RW}(t)$
- 4 **foreach** worker i **do**
- 5 $R_i \leftarrow |K \cap O_i| / |K|$
- 6 $L_i \leftarrow 1 - P_i / \max_j P_j$
- 7 $S_i \leftarrow 0.5 * R_i + 0.5 * L_i$
- 8 $i^* \leftarrow \arg \max_i S_i$ (break ties randomly)
- 9 Assign G to i^* ; update O_{i^*}

fall short of meeting all three at once. For uniform workloads, a naive load balancing policy (e.g., random) would suffice. The challenge lies in handling skewed workloads with hot objects. Leveraging the consensus window, we design a new policy, subgraph-first scheduling (SFS), that achieves all three objectives.

Our key insight is to leverage **subgraphs** as the unit of scheduling. For each batch of transactions from the consensus output, REMORA constructs a dependency graph (vertices are transactions and edges capture immediate dependencies induced by shared objects), identifies disjoint subgraphs, and dispatches those to the Workers. The intuition is that transactions within a subgraph already share objects and ordering constraints. Thus, collocating them on the same Worker avoids cross-node coordination. In practice, subgraphs typically arise from contention on hot objects, making them natural candidates for collocation.

REMORA avoids further partitioning such subgraphs [81], because it would not gain performance while introducing overhead. For extremely contended workloads, distributed stateful execution yields no benefits since the maximum concurrency is limited by the workload itself. Therefore, the contribution of this scheduling scheme is on low to medium level of contention. This idea aligns with prior work in distributed transactions: skewed, contended workloads benefit from single-node execution, while uniform workloads distribute effectively across nodes [62].

Algorithm 1 summarizes SFS. For each consensus batch, the Coordinator builds the dependency graph and schedules each disconnected subgraph G as a unit. It first computes the subgraph’s object footprint K as the union of the read/write sets $\text{RW}(t)$ of all transactions $t \in G$. Then, for each Worker i , SFS computes a *locality* score R_i as the fraction of K already owned by i (using the ownership map O_i), and a *load* score L_i from i ’s current load P_i . The final scheduling score S_i is the equal-weighted combination of locality and load (line 7). This jointly optimizes both, i.e., locality score reduces state movement and cross-Worker coordination, while load score prevents persistent skew and state accumulation on a single Worker [68, 69]. SFS assigns the entire subgraph to the Worker with the highest score and updates the ownership metadata accordingly. Stateless work is dispatched purely by load, which helps utilize idle Workers even when locality concentrates some stateful subgraphs.

4.6 Handling Failures

REMORA scales out single-validator execution and adopts the same failure model as existing blockchains that allows for validator crashes. In REMORA, we treat the failure of Coordinator, the sole externally visible interface to consensus, as equivalent to a single-validator failure. In that case, the remaining validators continue to make progress as long as the number of failed validators remains within the fault threshold of the underlying consensus protocol. To avoid reducing the validator’s MTTF due to the distributed architecture, we carefully design REMORA to tolerate Worker failures.

Periodic snapshotting. REMORA implements *periodic snapshotting* to keep the Coordinator loosely but consistently synchronized with the execution state in the Workers. Snapshotting operates in *epochs*. The Coordinator has full flexibility in determining the epoch size, choosing boundaries that either align with consensus rounds or adapt to workload characteristics, and explicitly notifies the Workers when an epoch ends. This exposes a tunable trade-off: longer epochs reduce snapshot overhead but increase recovery cost, while shorter epochs invert this trade-off. Workers keep track of the objects they modify within an epoch and once the epoch finishes, they send the Coordinator an *incremental* update containing only the latest versions of those modified objects. This incremental checkpointing on Workers amortizes frequent updates to hot objects without communicating every versioned update. With its global view of the system, the Coordinator knows exactly which object versions it needs at the end of each epoch. After collecting updates from all Workers, it atomically updates its state and advances `persist_index`, which marks the log position up to which all preceding transaction effects have been durably persisted.

Failure recovery. When a Worker fails, the Coordinator initiates recovery as follows. It first spawns a new Worker and determines the transaction *replay set*, defined as the range between the current dispatch index and the `persist_index`. From the log view, state up to `persist_index` is durable at the Coordinator. Thus, replaying after that point is required to reconstruct the parts of the missing suffix. The Coordinator reclaims ownership of all objects previously assigned to the failed Worker that are already durably persisted, and then dispatches the entire replay set to the new Worker. During replay, the Coordinator attaches required state that it has durably persisted, avoiding extra round-trip fetches. This procedure brings the new Worker online lazily, driven by transaction demand, to reduce stalls of excessive state transfers. After dispatching the replay set, the Coordinator can continue processing incoming transactions. We conservatively replay the entire replay set rather than attempting selective replay, as transactions may have been partially executed by either healthy or failed Workers and failures can occur at arbitrary points during execution. Notably, our design allows for skipping re-running all stateless parts as long as the Coordinator receives acknowledgement upon successful validation, which we leave for future work.

4.7 Elasticity

REMORA supports seamless elasticity by automatically scaling the Worker pool in response to workload fluctuations. The Coordinator orchestrates resource allocation by periodically monitoring per-Worker load and deciding when to spawn or retire Workers.

Crucially, autoscaling should not stall foreground execution or introduce any explicit rebalancing phase: REMORA leverages its lease-based ownership model to migrate objects across Workers and Coordinator dynamically and lazily.

Scale-out. The Coordinator spawns a new Worker and begins dispatching transactions to it. Initially, the new Worker has no local state. As it executes transactions, it fetches any objects it does not own from other Workers or from the Coordinator. Over time, state naturally migrates to the new Worker as leases are transferred during normal execution. Note that a new Worker can immediately start helping with stateless execution.

Scale-in. The Coordinator retires a Worker *gracefully* in two steps. First, it stops assigning new transactions to the retiring Worker, but keeps it online for a grace period to serve in-flight requests and inter-Worker lease transfers. Second, to complete retirement and reclaim ownership atomically, the Coordinator terminates the current snapshotting epoch for that Worker, forcing it to report all objects modified in that epoch back to the Coordinator. This yields an epoch-aligned handoff: hot objects migrate away naturally as their leases are transferred to other Workers, while cold objects are reclaimed by the Coordinator via the epoch snapshot where future transactions fetch them directly from the Coordinator.

5 IMPLEMENTATION

We implemented REMORA in 13k LoC of Rust. This includes a modular implementation of the Coordinator and Workers. We use tokio [97] for asynchronous network IO across nodes which communicate over TCP sockets.

Coordinator. Each module runs as a long-lived task pinned to a dedicated core. Modules on the critical transaction path (Figure 3) are connected by bounded channels, forming a pipeline. The router module spawns lightweight dispatching tasks to forward transactions according to the scheduling policy for stateful and stateless parts. These tasks run to completion in parallel on a thread pool, occupying all remaining cores, with one thread pinned per core. The autoscaler module periodically monitors the incoming load rate. Coordinator detects Worker failures by observing lost network connections and triggers the failure handling scheme (§4.6).

The Coordinator maintains metadata for each object and each node in the cluster. The object metadata takes the form of a map from object names to a tuple of (`current_version`, `current_owner`). The node metadata maintains per-Worker load information, which the stateless and stateful routing leverage to make load-aware decisions. The metadata footprint is modest: for 10M objects, the size of ownership metadata totals around 100 MB. This is in line with production chains; for example, although hundreds of millions of addresses have appeared on Ethereum, daily active addresses peak around 1.4M [30].

Worker. Each Worker node in REMORA runs a deterministic parallel runtime that handles both stateless and stateful compute. The runtime implements a custom asynchronous thread pool to drive the task execution. The design is agnostic to the underlying smart contract VM and focuses on parallel execution with strict ordering guarantees. Each execution unit, stateless or stateful, is an asynchronous task. Stateless tasks are immediately runnable and can run in parallel. Stateful tasks depend on (i) their corresponding stateless

tasks (e.g., authentication) and (ii) any previous tasks accessing the same objects. The runtime uses a dynamic DAG to keep track of the dependencies among tasks based on the specific object versions they access as assigned by the Coordinator. A certain task can only be executed when all its prior dependencies are satisfied.

The intra-Worker runtime implements the task DAG via `tokio::sync::Notify` [22], a lightweight primitive for synchronization. `Notify` carries no payload and is used purely for readiness signaling. Concretely, the runtime maintains one `Notify` instance per object version. When a transaction arrives with its required versions explicitly annotated, the runtime spawns an asynchronous task. This task (1) awaits on the `Notify` instances for all required input versions, ensuring all dependencies are satisfied; (2) once the dependencies become available executes the transaction to completion on the thread pool; (3) signals the `Notify` instances corresponding to the versions it produces, thereby unblocking dependent tasks. Remote dependencies work similarly, as when the Worker fetches the object, it signals the equivalent `Notify` instance. Finally, since each version is consumed exactly once, the runtime garbage-collects the corresponding `Notify` entry and the object version immediately after it has been consumed. Each `Notify` object has a unique name within the Worker and the runtime creates it atomically the first time it processes the producer or the consumer task. This design naturally integrates dependency enforcement into the asynchronous runtime, enabling deterministic parallel execution without any invasive changes to the existing scheduler, i.e., Tokio’s in the current implementation.

6 EVALUATION

Our evaluation aims to answer the following questions:

- Does REMORA’s *asymmetric* architecture improve efficiency? (§6.2)
- How does SFS compare against other distributed (non-)deterministic scheduling schemes? (§6.3)
- How much does the stateless-stateful separation improve performance? (§6.4)
- What is the benefit of leveraging the consensus window? (§6.5)
- How efficiently does REMORA recover from Worker failures? (§6.6)
- How does REMORA autoscale under varying loads? (§6.7)
- Can REMORA handle realistic dynamic workload hotspots? (§6.8)
- How scalable is REMORA’s Coordinator? (§6.9)

6.1 Experimental Setup

Testbed. We run our experiments on a 16-node cluster on AWS, using `m5d.8xlarge` instances in the same region. One node serves as the client node which generates transactions based on an open-loop Poisson process, one node serves as the Coordinator, and the others as Workers. Each machine is equipped with a 10Gbps NIC, 32 vCPUs (16 physical cores), and 128GB memory, which aligns with the recommended hardware configuration for typical validators [94–96]. We follow the layered approach of building blockchains [21]. REMORA is an execution layer system, hence our experiments focus on the execution layer of a single validator, which is the component REMORA changes. For the consensus layer, we implement a mock consensus module that adds a constant delay of 300ms equivalent

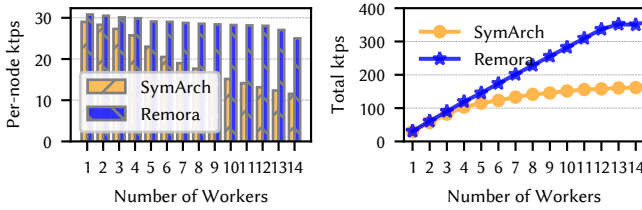


Figure 4: Comparison with other work using symmetric architecture.

to the consensus execution [8]. REMORA is agnostic to consensus deployment: variations such as co-located vs. geo-distributed validators, heterogeneous capacities, or Byzantine behavior would manifest only as variability in the consensus window duration. Shorter windows would reduce the slack for pre-consensus optimizations and longer windows would strictly increase these benefits.

Workloads. Following the methodology of prior work [20, 69, 92], we evaluate REMORA using YCSB [23], TPC-C [1], and Ethereum mainnet traces [86]. For YCSB, we implement a smart contract with a global mapping that represents a key-value store over a 10M-key keyspace, where each object is modeled as a 0.25 kB account. To study contention and skew, we vary the number of keys accessed per transaction under both uniform and Zipfian distributions. Each transaction contains a fixed number of read-modify-write operations, each on a unique key. Since REMORA is VM-agnostic, we factor out VM-specific effects by modeling execution as a stateless stage followed by a stateful stage. Because real smart-contract VMs incur high overhead while pure YCSB reads/writes are lightweight, we add synthetic spinning to emulate realistic execution. Unless otherwise stated, YCSB uses 0.5 ms synthetic time for both stages, consistent with §2 and prior measurements on common smart contracts [102]. We also validate real execution by running the same experiments on SuiVM and real-world cryptographic functions (§6.4). For TPC-C, we implement a smart contract supporting New-Order and Payment transactions, which account for 88% of the workload and involve remote data access [68], with 40 warehouses per Worker. Since TPC-C is already heavier than YCSB, its stateful stage runs the real business logic, while the stateless stage uses the same 0.5 ms synthetic cost.

6.2 Architectural Efficiency

Methodology. We start by comparing REMORA’s *asymmetric* architecture with a single node in charge of scheduling to prior Calvin-inspired [99] *symmetric* schemes [69, 78, 80, 85, 92] where every node accepts and deterministically schedules all transactions. To evaluate the efficiency of these two choices, we implement a baseline, SymArch, where each Worker performs both deterministic scheduling and execution to represent this line of work. Since sequencing in our setting is established via consensus, we exclude sequencing from both systems to ensure a fair comparison. We use the YCSB benchmark with uniform distribution and a round-robin scheduling policy instead of SFS, while disabling the stateless-stateful separation. In this configuration, the ideal per-Worker throughput is about 32k TPS, given 32 vCPUs and a 1 ms per-transaction service time. We vary the number of Workers and measure the maximum throughput.

Results. Figure 4 shows that REMORA delivers higher throughput per deployed node than SymArch as the number of Workers

increases. In SymArch, every Worker repeatedly pays the cost of deterministic scheduling, consuming CPU cycles that could otherwise execute transactions. This redundant overhead directly reduces the useful work obtained from each provisioned machine. In contrast, REMORA offloads all scheduling to the Coordinator, allowing Workers to dedicate their full capacity to transaction execution, improving resource efficiency and translating the same hardware budget into more throughput. Although REMORA requires an additional coordination node, the net deployment cost is still lower for a target throughput: for example, a 4-worker REMORA configuration already outperforms a 5-worker SymArch. We expect the efficiency gap to widen further with more advanced scheduling policies that are more computationally expensive compared to round robin. REMORA’s near-linear scaling stops once the Coordinator becomes the bottleneck around 340 kTPS, which we explain in §6.9. Notably, REMORA can also support more Workers when smart contract is compute-heavier, or when Worker instance is smaller.

6.3 Scheduling Schemes Performance

Methodology. We compare SFS against several state-of-the-art scheduling schemes, including both deterministic and non-deterministic designs by measuring the max achieved throughput. **Hermes** [69] represents the state-of-the-art for deterministic scheduling. It considers both load and locality but violates strict determinism, since it relies on transaction reordering to improve performance. We also implement a strictly-deterministic variant, **Hermes (NR)**, that disables reordering in Hermes. **RTSFaaS** [122] is another scheme that considers both locality and load balance initially designed for FaaS workloads, yet is not deterministic. **Zeus** [50] is a locality-only baseline that aggressively migrates objects to the Worker executing each distributed transaction. We also include simple load-balancing policies: RoundRobin and Random. Notably, under uniform or low-skew workloads, all policies achieve similar performance as transactions are evenly distributed and require minimal object migration. This experiment thus focuses on contended workloads. For an apples-to-apples comparison of scheduling policies, we disable REMORA’s use of the consensus window and the stateless-stateful separation.

YCSB. As shown in Figure 5, naive policies perform the worst due to frequent distributed transactions and object transfers. Zeus ignores load imbalance and gradually migrates all objects to a single Worker. After warm-up, only one Worker is busy while others remain idle, which is consistent with prior observations [69]. However, under high contention (3 or 5 objects with $\alpha = 0.9$) in which the concurrency level in the workload is extremely limited, the benefits of scale-out diminish and single-node execution outperforms others. RTSFaaS and Hermes (NR) achieve similar performance (11% to 19% improvement over round-robin) by considering both load and locality. The full Hermes policy benefits further from reordering up to 21%, yielding the best baseline performance.

SFS consistently outperforms all baselines (up to 28%), including non-deterministic ones, across all contended configurations. Whereas Hermes and RTSFaaS balance load by migrating individual transactions, SFS schedules at the subgraph level, colocating dependent and locality-sharing transactions. This granularity is dependency-aware and minimizes remote state transfer, confirming

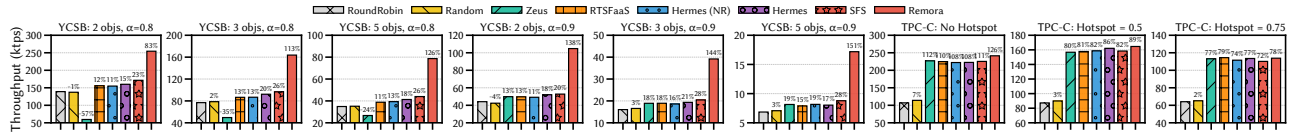


Figure 5: Performance comparison with other policies. Percentage denotes the improvement over round-robin policy.

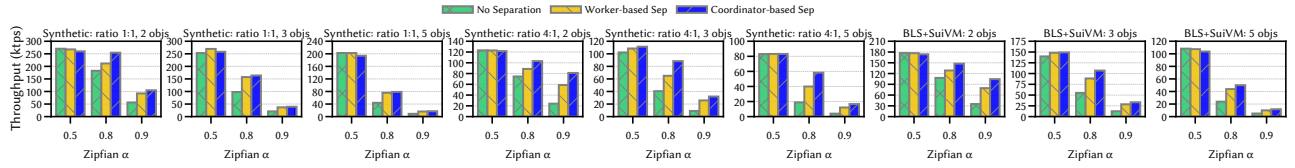


Figure 6: Benefits of stateless-stateful separation. Ratio denotes the duration ratio of stateless and stateful part in each transaction.

that subgraphs are the right unit for scale-out scheduling. We also include REMORA, which is the SFS policy with the optimizations of stateless-stateful separation and consensus window execution.

TPCC. For TPC-C benchmarks, we pre-partition warehouses among Workers, and construct the hotspot by setting 50% and 75% of transactions targeting at the warehouses pre-assigned on the same Worker. SFS does not yield noticeable throughput gains over the baselines on TPC-C. This is expected because TPC-C uses warehouse-based partitioning by design, and most transaction state accesses are by the home warehouse/district while cross-warehouse accesses are relatively limited (default 15% in NewOrder and Payment transactions). Consequently, under TPC-C the bottleneck is dominated by warehouse-level load skew rather than distributed execution, so different locality-aware policies converge in performance. Our observation is consistent with prior work [69, 92].

Lease transfer analysis. We measure the network consumption of Workers involved in object migration and report their overhead. Under uniform YCSB cases where total throughput reaches 100 k TPS, lease transfer with two objects per transaction consumes 24 MiB/s of network bandwidth, of which 73% is inbound due to transactions received from Coordinator. In contended cases where $\alpha = 0.9$, the bandwidth consumed by the most heavily loaded Worker increases to 33 MiB/s. Bandwidth consumption also rises with the number of objects per transaction. For example, with five objects per transaction, the consumed bandwidth increases to 62 MiB/s. Overall, this level of network consumption does not pose a deployment concern for modern cloud infrastructures.

6.4 Impact of Stateless-Stateful Separation

Methodology. We study the benefits of separating the stateless from the stateful part of a transaction in the achieved throughput. We consider two separation modes: (i) *Worker-based Sep*, where only Workers apply stateless-stateful separation during execution. This configuration could even be applied in single-node runtimes or distributed runtimes without centralized scheduling. (ii) *Coordinator-based Sep*, in which the Coordinator can schedule stateless and stateful parts on different Workers. We use the SFS scheduling policy and the YCSB benchmark to study such impact.

Results. Figure 6 shows that *Worker-based Sep* improves throughput by up to 2 $\times$, as separating execution at Workers unlocks greater parallelism across available cores. *Coordinator-based Sep* achieves up to 3 $\times$ improvement, with the most pronounced advantage when

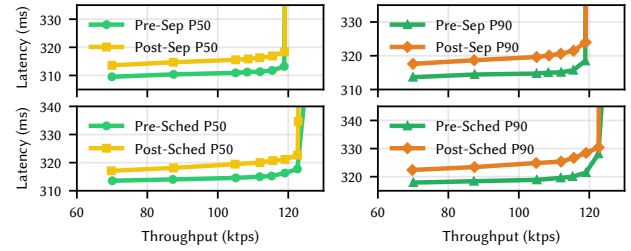


Figure 7: Benefits of consensus window. *Pre-Sep* denotes pre-consensus stateless execution, while *Post-Sep* denotes post-consensus stateless execution; *Pre-Sched* denotes pre-consensus scheduling, while *Post-Sched* denotes post-consensus scheduling.

the stateless dominates. These results highlight that under skewed workloads, separation at the Coordinator effectively harnesses otherwise idle compute capacity across Workers. In contrast, under low contention the benefits diminish, as load is balanced.

6.5 Benefits of Consensus Window

Methodology. REMORA leverages the consensus window to perform stateless execution and stateful scheduling. We study the effect of this design choice on latency. Given that the benefits are independent of scheduling schemes, we use the uniform YCSB benchmark, 2ms of stateless, and 0.5ms of stateful processing, and SFS scheduling. As a baseline, we disable pre-consensus scheduling and decoupled stateless execution and measure the end-to-end latency with achieved throughput.

Pre-consensus stateless execution. In Figure 7, *Pre-Sep* demonstrates clear latency improvement by overlapping stateless execution with the consensus window. *Post-Sep* incurs an additional 2–5ms at both P50 and P90 latency. Given that the stateless portion must always be executed and is fully parallelizable, the performance gain from pre-consensus execution is guaranteed, and becomes even more pronounced as the stateless workload grows.

Pre-consensus stateful scheduling. A similar effect arises when scheduling is shifted into the consensus window. To isolate this effect, we keep stateless execution in post-consensus for this experiment. Although our scheduling policy is lightweight (~4ms per batch), latency reduction of *Pre-Sched* indicates that the consensus window provides sufficient slack to accommodate more sophisticated scheduling strategies off-path, e.g., ML-based ones.

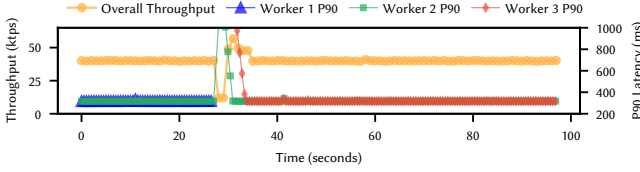


Figure 8: Failure recovery. Worker 1 failed and Worker 3 is spawned.

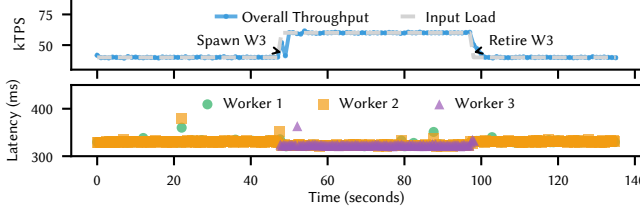


Figure 9: Elastic scaling for load fluctuations.

6.6 Failure Recovery

We evaluate REMORA’s failure recovery by subjecting it to a constant YCSB uniform workload while injecting a Worker failure at a random time. From this experiment onward, we use a smaller 3-Worker pool, as these experiments focus on demonstrating REMORA’s mechanisms and the feasibility of properties rather than its performance.

Figure 8 illustrates the recovery process. Initially, Worker 1 and Worker 2 jointly handle the incoming requests. Upon detecting the failure of Worker 1 at $t = 28s$, REMORA immediately switches on Worker 3 as a replacement. We omit the cost of provisioning an AWS EC2 VM and instead pre-launch an idle Worker before the experiment. During recovery, the system experiences a brief throughput degradation of approximately 2 seconds while it determines the replay set, transfers object ownership, and dispatches the replay set to Worker 3. Subsequently, Worker 2 and Worker 3 enter a catch-up phase, processing requests at their maximum throughput capacity, which temporarily elevates latency. Once the backlog is cleared, the performance stabilizes at the pre-failure levels, demonstrating REMORA’s resilience to Worker failures.

6.7 Elasticity

We study REMORA’s elasticity capabilities by applying load that changes over time via uniform YCSB workloads. Figure 9 shows the input load pattern: throughput increases sharply from 40k TPS to 60k TPS and then decreases to 40k TPS. Once the autoscaler detects the load spike which exceeds the processing capacity of existing Workers, it starts a new Worker instance (pre-launched as same as the failure experiment). The Coordinator immediately begins forwarding load to the new Worker, which incrementally fetches required state from existing nodes on demand. When the load drops to 40k TPS, the autoscaler initiates scale-in and gracefully retires Worker 3. Across both transitions, latency remains low and stable, showing that REMORA can autoscale without stalling foreground execution or triggering an explicit rebalancing phase.

6.8 Handling Dynamic Hotspots

We evaluate REMORA’s ability to handle dynamic hotspots, i.e., object popularity changes, via both real-world traces from the Ethereum mainnet [86] and TPC-C benchmarks. For the Ethereum traces, we derive object access distributions from different blocks

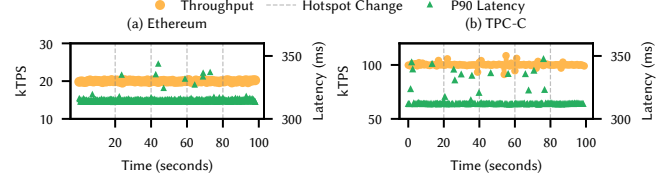


Figure 10: Dynamic hotspot workloads.

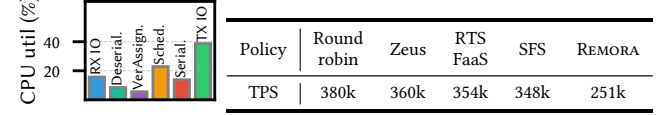


Figure 11: Coordinator analysis. The left figure shows the CPU utilization breakdown, while the right table shows the max throughput.

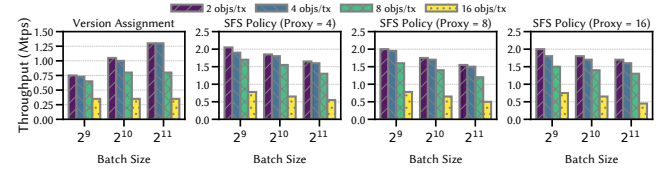


Figure 12: VerAssign. and Sched. (SFS) module sensitivity analysis.

spanning different periods of time to capture realistic contention. We run this workload on a production-grade smart contract VM, SuiVM [59]. For the TPC-C experiment, we target 50% of the transactions to warehouses on a single Worker node to emulate hotspots. The workload changes every 20s, causing hotspots to shift dynamically across Workers and mimicking temporal locality observed in practice. We set the input rate to 80% of the measured system capacity for each workload (accounting for available parallelism and VM overhead) based on offline profiling. REMORA shows only occasional high-latency outliers (Figure 10).

6.9 Scalability

While REMORA’s asymmetric architecture enables near-linear scale-out with the number of Workers (§6.2), centralizing transaction dispatch at the Coordinator raises the concern that it could become a throughput bottleneck. We therefore stress-test the Coordinator under a range of scheduling policies. To isolate dispatching overhead, we use uniform distribution with zero service time, ensuring that Workers do not limit throughput. As shown in Figure 11, with a simple round-robin policy the system sustains nearly 380k TPS. Adding computation for locality calculation and dependency analysis reduces throughput only slightly, to around 350k TPS. Separating stateless and stateful execution reduces throughput to 250k TPS due to the extra message serialization, I/O, and syscall overhead. Given that state-of-the-art consensus modules achieve 200~300k TPS [7, 8], we conclude that the Coordinator in our design is sufficient for practical deployment.

To dive deeper into the Coordinator’s design, Figure 11 also reports the per-module CPU utilization breakdown per 1k transactions. Notably, when removing stateless-stateful separation, the TX-IO part only takes 25%, demonstrating the extra overhead of the separation. Leveraging syscall batching and kernel bypass could mitigate this, which we leave for future work. We also observe that all cores are saturated, indicating that larger EC2 instances would

yield proportionally higher throughput, as the network tasks are parallelizable.

In all these modules, version assignment is the main *serial* component, making it the primary scalability bottleneck as per Amdahl’s Law. We also study each module in isolation to identify which parameters affect performance. We find that the number of objects per transaction is the key limiting factor for version assignment. In the scheduling module, though building dependency graphs can be parallelized, the locality-related part is sequential as to know the scheduling decision made for overlapped objects. As shown in Figure 12, we sweep the number of Workers, batch size, and objects per transaction, and find that SFS’s throughput is mainly sensitive to the latter two, due to dependency-graph construction and subgraph assignment overhead.

7 DISCUSSION

Read/write set assumption. REMORA relies on knowing each transaction’s read/write set before stateful execution. This requirement restricts expressiveness: it does not directly support contracts whose accessed state is discovered only during execution. Thus, REMORA targets execution environments where access sets are part of the transaction interface, can be conservatively derived before execution, or are provided by clients/builders through off-chain simulation. Such environments still cover many common smart-contract workloads, including token transfers, decentralized exchanges, and NFT mints. This model is common in deterministic database systems [32, 33, 42, 71, 78, 80, 82, 85, 99] and research blockchain systems [4, 19, 81], and has been shown to hold for a majority of transactional workloads in a recent workload-characterization study [77]. Some prior systems based on optimistic execution [35, 61, 73] relax this limitation, but at the cost of violating strict determinism. Recent work [6, 31] further shows that access specifications can benefit optimistic execution by reducing aborts. This constraint simplifies our enforcement of strict determinism and enables efficient scheduling with locality awareness. Furthermore, it is also a practical design point in several production blockchains: Solana and Sui expose transaction-level account/object access metadata for parallel execution [90, 93], while Ethereum is set to enforce block-level access lists [101] as a headliner feature in its upcoming Glamsterdam upgrade [29]. Thus, REMORA can be directly integrated into any existing blockchain system with a modular execution layer whose access set can be determined before execution.

Scaling L1 on-chain compute. Emerging on-chain workloads (AI inference, ZK proofs, and post-quantum crypto [3, 26, 57, 103]), are compute-heavy and currently pushed to L2s or off-chain due to incapability of current L1 infrastructure. REMORA removes this reliance via elastic scale-out, improving L1 performance while avoiding fragmentation and enabling new L1-native use cases.

DoS resilience. Stateless validation often involves expensive cryptographic checks, making it a common vector for DoS attacks. REMORA mitigates this risk by elastically scaling stateless execution across nodes, absorbing heavy or adversarial workloads without introducing coordination bottlenecks. This elasticity ensures system responsiveness and robustness even under targeted load. As blockchains adopt quantum-safe cryptography and other compute-intensive primitives, such resilience becomes increasingly essential.

8 RELATED WORK

Deterministic Distributed Transactions. Most scale-out deterministic transactional systems adopt Calvin [99]’s *symmetric* architecture, unlike REMORA’s asymmetric design. Some focus on improving the sequencing layer, while others optimize the partitioning and scheduling scheme. SLOG [85] and Detock [78] decentralize the sequencer to support geo-distributed deployments. Caerus [42] minimizes the round-trip traversal by deterministically merging per-region partial sequences using a new ordering protocol. Q-Store [80] bypasses the single-threaded sequencing and scheduling parts via constructing fine-grained execution queues.

Deterministic Single-Node Transactions. Deterministic parallelism has been widely studied in databases and systems. A broad line of work follows a pessimistic *schedule-then-execute* pattern, with mechanisms spanning dependency-graph analysis [33, 43, 53, 71, 73], multi-versioning [32, 82], and per-object queues [16]. REMORA similarly decouples scheduling from execution. Other work adopts optimistic *execute-then-validate* schemes [49, 73], which avoid pre-declared read/write sets but can incur aborts under contention and relax strict adherence to the agreed-upon order.

Non-Deterministic Distributed Transactions. This line of work relies on variants of two-phase commits (2PC) to coordinate multi-partition transactions. Some accelerate 2PC using new hardware technologies such as RDMA [27, 116] and CXL [46, 47] to reduce communication costs. Eris [64] exploits programmable switches for transaction sequencing. Centralized transaction routing with ownership-based data migration is also adopted in several shared-storage databases [45, 63]. GaussDB [63] employs a centralized routing model performing inference while deploying training elsewhere to avoid the on-path overhead. Our proposed consensus window can allow such computationally expensive and ML-based routing schemes. Pegasus [65] uses an in-network directory to track objects location, similar to how Coordinator tracks ownership. Lotus [123] optimizes multi-partition transactions by introducing granule locks to mitigate distributed commit overhead. Besides, epoch-based commit/recovery and asynchronous persistence in prior work [74, 119] are similar to our design.

Smart Contract Execution. Existing efforts primarily exploit multi-core parallelism for deterministic execution. Recent work targets finer-grained operation-level optimizations to mitigate rollback overhead under optimistic concurrency control [20, 35, 67, 100, 107]. OptME [87] builds key-based dependency graphs to derive parallel execution schedules while extensively using re-ordering. Some other work leverages new programming semantics to enhance performance, including commutativity [84, 102] and deferred objects [76]. Spectrum [20] carefully designs rollback and re-execution mechanisms to support unknown read/write sets while preserving strict determinism. Pre-execution is also used to synthesize accelerated EVM code by caching speculative results for all possible branches [18, 117]. DeCl [111] adopts software-based sandboxing to perform deterministic gas metering on untrusted machine code. In Vegeta [110], each validator speculatively executes its proposed blocks to generate hints, and after consensus all validators replay all blocks. These mechanisms are orthogonal to the design of REMORA.

Blockchain Architectures. AdaptChain [109] scales throughput by adjusting concurrent block generation based on demand. Some

systems move state off-chain to increase throughput [17, 108]. Pilotfish [51] adopts the Calvin architecture and uses fixed data partitioning, thus inelastic and inadaptive to workloads. In permissioned systems, SChain [81] adopts a similar architecture in its single trusted organization and decouples execution into compute and storage roles. FlexChain [104] leverages hardware disaggregation to improve resource utilization. REMORA differs in leveraging domain insights and focusing on the performance and operational practicality of a modular execution layer, instead of the whole chain.

9 CONCLUSION

In this work, we make the case for a scale-out blockchain execution layer that preserves strict determinism. We show why prior deterministic transactional systems fall short and identify opportunities unique to blockchains. Based on these insights, we design REMORA, a scale-out smart contract execution engine. REMORA adopts an asymmetric architecture for efficiency and cost-effectiveness, and employs versioning with ownership to guarantee determinism. REMORA separates the stateless part from smart contract transactions and leverages consensus window to perform stateless execution and stateful scheduling. REMORA also demonstrates its elasticity and workload adaptiveness. We believe REMORA paves the way for the next generation of scalable blockchain systems.

REFERENCES

- [1] 2023. TPC-C Benchmark. <https://www.tpc.org/tpcc/>. Accessed: December 2, 2023.
- [2] Michael Abebe, Brad Glasbergen, and Khuzaime Daudjee. 2020. MorphoSys: automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.
- [3] Algorand Foundation. 2025. Leading on post-quantum technology. <https://algorand.co/technology/post-quantum>.
- [4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Par-blockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.
- [5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 international conference on management of data*. 76–88.
- [6] Parwat Singh Anjana, Matin Amini, Rohit Kapoor, Rahul Parmar, Raghavendra Ramesh, Srivatsan Ravi, and Joshua Tobkin. 2025. Efficient Parallel Execution of Blockchain Transactions Leveraging Conflict Specifications. In *7th Conference on Advances in Financial Technologies (AFT 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 29–1.
- [7] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2025. Shoal+: High Throughput {DAG} {BFT} Can Be Fast and Robust!. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 813–826.
- [8] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2023. Mysticeti: Reaching the limits of latency with uncertified dags. *arXiv preprint arXiv:2310.14821* (2023).
- [9] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindström, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. 2024. zklogin: Privacy-preserving blockchain authentication with existing credentials. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. 3182–3196.
- [10] Dvir David Biton, Roy Friedman, and Yaron Hay. 2025. Ethereum Conflicts Graphed. *arXiv:2507.20196 [cs.DC]* <https://arxiv.org/abs/2507.20196>
- [11] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Salt Lake City, UT, USA, 2606–2620. <https://doi.org/10.1145/3658644.3670286>
- [12] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptography* 17, 4 (2004), 297–319.
- [13] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. 2021. The provable security of ed25519: theory and practice. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1659–1676.
- [14] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association, 173–186.
- [15] Kostas Kryptos Chalkias, Jonas Lindström, Deepak Maram, Ben Riva, Arnab Roy, Alberto Sonnino, and Joy Wang. 2024. Fastcrypto: Pioneering cryptography via continuous benchmarking. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*. 227–234.
- [16] Chen Chen, Xingbo Wu, Wenshao Zhong, and Jakob Eriksson. 2024. Fast Abort-Freedom for Deterministic Transactions. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 692–704.
- [17] Wuhui Chen, Ding Xia, Zhongteng Cai, Hong-Ning Dai, Jianting Zhang, Zicong Hong, Junyuan Liang, and Zibin Zheng. 2024. Polygon: Scaling blockchain via 3d parallelism. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1944–1957.
- [18] Yang Chen, Zhongxin Guo, Runhui 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.
- [19] Zhihao Chen, Xiaodong Qi, Xiaofan Du, Zhao Zhang, and Cheqing Jin. 2021. Peep: A parallel execution engine for permissioned blockchain systems. In *International Conference on Database Systems for Advanced Applications*. Springer, 341–357.
- [20] Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2541–2554.
- [21] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Proof of Availability & Retrieval in a Modular Blockchain Architecture. *Cryptology ePrint Archive*, Paper 2022/455. <https://eprint.iacr.org/2022/455>
- [22] Tokio Contributors. [n.d.]. Notify in tokio::sync. Rust crate documentation on docs.rs. <https://docs.rs/tokio/latest/tokio/sync/struct.Notify.html> Accessed: 2025-12-31.
- [23] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [24] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).
- [25] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.
- [26] DFINITY Foundation. 2024. The Next Step for DeAI: On-Chain Inference Enabling Face Recognition. <https://medium.com/dfinity/the-next-step-for-deai-on-chain-inference-enabling-face-recognition-589183203fc2>. DFINITY Blog.
- [27] Tamer Eldeeb, Xincheng Xie, Philip A Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and general datacenter transactions for {On-Disk} databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 343–360.
- [28] Ethereum.org. 2025. *Ethereum roadmap*. <https://ethereum.org/roadmap/> Accessed: 2026-01-14.
- [29] Ethereum.org. 2026. *Glamsterdam*. <https://ethereum.org/roadmap/glamsterdam/#bals> Page last updated: April 13, 2026. Accessed: May 22, 2026.
- [30] Etherscan. [n.d.]. Ethereum Active Addresses Chart. Etherscan. <https://etherscan.io/chart/active-address> Accessed: 2025-12-31.
- [31] François Ezard, Can Umut Ileri, and Jérémie Decouchant. 2025. NEMO: Faster Parallel Execution for Highly Contended Blockchain Workloads. In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 1–5.
- [32] Jose M Faleiro and Daniel J Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.
- [33] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).
- [34] FastCrypto. 2025. FastCrypto Benchmark. <https://mystenlabs.github.io/fastcrypto/benchmarks/criterion/reports/BN254%20Verification/Groth16%20verify%20with%20processed%20vk/8/index.html>. Accessed: July 22, 2025.
- [35] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. 2022. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *Proceedings of the 44th international conference on software engineering*. 2315–2326.
- [36] 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 *PPoPP*. ACM, 232–244.
- [37] Sara Gherghelas. 2024. DappRadar Games Report – 2023 Overview. <https://dappradar.com/blog/dappradar-games-report-2023-overview>. DappRadar × BGA Blog.
- [38] Paul Grau. [n.d.]. Lessons learned from making a Chess game for Ethereum. <https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6>. Accessed: 2025-09-08.
- [39] Yaron Hay and Roy Friedman. 2024. Batch-Schedule-Execute: On Optimizing Concurrent Deterministic Scheduling for Blockchains. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 163–174.
- [40] Lioba Heimbach, Quentin Knip, Yann Vonlanthen, and Roger Wattenhofer. 2023. Defi and nfts hinder blockchain scalability. In *International conference on financial cryptography and data security*. Springer, 291–309.
- [41] Jelle Hellings and Mohammad Sadoghi. 2021. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2230–2243.
- [42] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proceedings of the VLDB Endowment* 17, 3 (2023), 469–482.
- [43] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. 2013. KuaFu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1186–1195.
- [44] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. 2024. GridDB: Scaling blockchain database via sharding and off-chain cross-shard mechanism. *arXiv preprint arXiv:2407.03750* (2024).
- [45] Chunyue Huang, Shuang Liu, Xinyi Zhang, Wenhao Li, Wei Lu, and Xiaoyong Du. 2025. Chimera: Mitigating Ownership Transfers in Multi-Primary Shared-Storage Cloud-Native Databases. *Proceedings of the VLDB Endowment (PVLDB)* 18, 10 (2025), 3368–3381. <https://doi.org/10.14778/3748191.3748201>
- [46] Yibo Huang, Haowei Chen, Newton Ni, Vijay Chidambaram, Dixin Tang, Emmett Witchel, Zhiting Zhu, and Zhipeng Jia. 2025. Tigon: A distributed database for a CXL pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston, MA.
- [47] Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. 2025. Pasha: An efficient, scalable database architecture for cxl pods. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.
- [48] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.
- [49] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about eve: {Execute-Verify} replication for {Multi-Core} servers. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 237–250.
- [50] Antonios Katsarakis, Yijun Ma, ZhaoWei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 145–161.
- [51] Quentin Knip, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. 2024. Pilotfish: Distributed transaction execution for lazy blockchains. *arXiv preprint arXiv:2401.16292* (2024).
- [52] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.
- [53] Ramakrishna Kotla and Michael Dahlin. 2004. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*. IEEE, 575–584.
- [54] Aptos Labs. 2025. Aptos. <https://aptoslabs.com/>. Accessed: July 22, 2025.
- [55] Aptos Labs. 2025. Aptos Code Repository. <https://github.com/AptosLabs/apto-core>. Accessed: July 22, 2025.
- [56] Cronos Labs. [n.d.]. Cronos. <https://cronos.org/>. Accessed: 2025-09-05.
- [57] Mysten Labs. 2025. FastCrypto GitHub Repository. <https://github.com/MystenLabs/fastcrypto>. Accessed: July 22, 2025.
- [58] Mysten Labs. 2025. Sui. <https://sui.io/>. Accessed: July 22, 2025.
- [59] Mysten Labs. 2025. Sui Code Repository. <https://github.com/MystenLabs/sui>. Accessed: July 22, 2025.
- [60] Solana Labs. 2025. Solana Code Repository. <https://github.com/solana-labs/solana>. Accessed: July 22, 2025.
- [61] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When private blockchain meets deterministic database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–28.
- [62] Jennifer Lam, Jeffrey Helt, Wyatt Lloyd, and Haonan Lu. 2024. Accelerating Skewed Workloads With Performance Multipliers in the {TurboDB} Distributed Database. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1213–1228.
- [63] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.
- [64] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 104–120.
- [65] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 387–406.
- [66] Zhuolun Li, Alberto Sonnino, and Philipp Jovanovic. 2023. Performance of edds and bls signatures in committee-based consensus. In *Proceedings of the 5th workshop on Advanced tools, programming languages, and Platforms for Implementing and Evaluating algorithms for Distributed systems*. 1–5.
- [67] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. 2025. ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems*. 211–225.
- [68] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data*. 1659–1674.
- [69] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don’t look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1156–1168.
- [70] Yushi Liu, Liwei Yuan, Zhihao Chen, Yekai Yu, Zhao Zhang, Cheqing Jin, and Ying Yan. 2023. ChainDash: An Ad-Hoc Blockchain Data Analytics System. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4022–4025.
- [71] Zhengqing Liu, Musa Unal, Matthew J Parkinson, and Marios Kogias. 2025. DORADD: Deterministic Parallel Execution in the Era of Microsecond-Scale Computing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 282–296.
- [72] Yuen C Lo and Francesca Medda. 2021. Uniswap and the Emergence of the Decentralized Exchange. *Journal of financial market infrastructures* 10, 2 (2021), 1–25.
- [73] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. (2020).
- [74] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14, 5 (2021), 743–756. <https://doi.org/10.14778/3446095.3446098>
- [75] Bhavana Mehta, Nupur Baghel, Mohammad Javad Amiri, Boon Thau Loo, and Ryan Marcus. 2025. Adaptive Sharding in Untrusted Environments. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–28.
- [76] George Mitenkov, Igor Kabiljo, Zekun Li, Alexander Spiegelman, Satyanarayana Vusirikala, Zhuolun Xiang, Aleksandar Zlateski, Nuno P Lopes, and Rati Gelashvili. 2024. Deferred objects to enhance smart contract programming with optimistic parallel execution. *arXiv preprint arXiv:2405.06117* (2024).
- [77] Cuong DT Nguyen, Kevin Chen, Christopher DeCarolis, and Daniel J Abadi. 2025. Are Database System Researchers Making Correct Assumptions about Transaction Workloads? *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–26.
- [78] Cuong DT Nguyen, Johann K Miller, and Daniel J Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.
- [79] Okta, Inc. 2024. What is Decentralized Identity? <https://www.okta.com/blog/identity-security/what-is-decentralized-identity/>. Okta Blog.
- [80] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *EDBT*. 73–84.
- [81] Xiaodong Qi, Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2023. Schain: Scalable concurrency over flexible permissioned blockchain. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1901–1913.
- [82] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.
- [83] Kaihua Qin, Stefanos Chaliasos, Liyi Zhou, Benjamin Livshits, Dawn Song, and Arthur Gervais. 2023. The blockchain imitation game. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3961–3978.
- [84] Geoffrey Ramseyer and David Mazières. 2024. Groundhog: Linearly-scalable smart contracting via commutative transaction semantics. *arXiv preprint arXiv:2404.03201* (2024).
- [85] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019).
- [86] risechain. 2025. risechain/pevm: Blazingly fast Parallel EVM. <https://github.com/risechain/pevm>. Accessed: 2025-09-05.
- [87] Donghyeon Ryu and Chanik Park. 2024. 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. IEEE, 1–16.

- [88] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.
- [89] Zeshun Shi, Cees De Laat, Paola Grosso, and Zhiming Zhao. 2022. Integration of blockchain and auction models: A survey, some applications, and challenges. *IEEE Communications Surveys & Tutorials* 25, 1 (2022), 497–537.
- [90] Solana Foundation. 2026. Transaction Structure. <https://solana.com/docs/core/transactions/transaction-structure>. Solana Documentation. Accessed: 2026-05-12.
- [91] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.
- [92] Yuan Sui, Xiaochun Yang, Bin Wang, Yujie Zhang, and Baihua Zheng. 2025. Wait and See: A Delayed Transactions Partitioning Approach in Deterministic Database Systems for Better Performance. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 1–27.
- [93] Sui Foundation. 2026. Building Transactions. <https://docs.sui.io/develop/transactions/ptbs/building-ptb>. Sui Documentation. Accessed: 2026-05-12.
- [94] The Aptos Team. 2025. Validator Deployment and Configuration. <https://aptos.dev/network/nodes/validator-node/node-requirements>.
- [95] The Sui Team. 2025. Validator Deployment and Configuration. <https://docs.sui.io/guides/operator/validator/validator-config>.
- [96] The Solana Team. 2025. Validator Deployment and Configuration. <https://docs.solanalabs.com/operations/requirements>.
- [97] The Tokio Team. [n.d.]. Tokio: an asynchronous Rust runtime. <https://tokio.rs/>. Accessed: 2025-09-08.
- [98] Justin Thaler. 2025. Quantum computing and blockchains: Matching urgency to actual threats. a16z crypto. <https://a16zcrypto.com/posts/article/quantum-computing-misconceptions-realities-blockchains-planning-migrations/>. Accessed 2026-01-13.
- [99] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.
- [100] Xing Tong, Zheming Ye, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. TELL: Efficient Transaction Execution Protocol Towards Leaderless Consensus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1902–1915.
- [101] Toni Wahrstätter, Dankrad Feist, Francesco D’Amato, Jochem Brouwer, Ignacio Hagopian, Felipe Selmo, Rahul, and Stefan. 2025. EIP-7928: Block-Level Access Lists. <https://eips.ethereum.org/EIPS/eip-7928>. Ethereum Improvement Proposals, Draft. Accessed: 2026-05-12.
- [102] Hao Wang, Minghao Pan, and Jiaping Wang. 2025. Crystallinity: A Programming Model for Smart Contracts on Parallel EVMs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 412–425.
- [103] Zhipeng Wang, Rui Sun, Elizabeth Lui, Tuo Zhou, Yizhe Wen, and Jiahao Sun. 2025. AIArena: A blockchain-based decentralized AI training platform. In *Companion Proceedings of the ACM on Web Conference 2025*. 1375–1379.
- [104] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. 2022. FlexChain: an elastic disaggregated blockchain. *Proceedings of the VLDB Endowment* 16, 1 (2022).
- [105] Chenyuan Wu, Mohammad Javad Amiri, Haoyun Qin, Bhavana Mehta, Ryan Marcus, and Boon Thau Loo. 2024. Towards Full Stack Adaptivity in Permissioned Blockchains. *Proceedings of the VLDB Endowment* 17, 5 (2024).
- [106] Chenyuan Wu, Bhavana Mehta, Mohammad Javad Amiri, Ryan Marcus, and Boon Thau Loo. 2023. AdaChain: A Learned Adaptive Blockchain. *Proceedings of the VLDB Endowment* 16, 8 (2023), 2033–2046.
- [107] Jiang Xiao, Shijie Zhang, Zhiwei Zhang, Bo Li, Xiaohai Dai, and Hai Jin. 2022. Nezha: Exploiting concurrency for transaction processing in dag-based blockchains. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 269–279.
- [108] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2314–2326.
- [109] Jie Xu, Qingyuan Xie, Sen Peng, Cong Wang, and Xiaohua Jia. 2023. Adaptchain: Adaptive scaling blockchain with transaction deduplication. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1909–1922.
- [110] Tianjing Xu, Yongqi Zhong, Yiming Zhang, Ruofan Xiong, Jingjing Zhang, Guangtao Xue, and Shengyun Liu. 2025. Vegeta: Enabling Parallel Smart Contract Execution in Leaderless Blockchains. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 795–811. <https://www.usenix.org/conference/nsdi25/presentation/xu-tianjing>
- [111] Zachary Yedidia, Geoffrey Ramseyer, and David Mazières. 2025. Deterministic Client: Enforcing Determinism on Untrusted Machine Code. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 633–649.
- [112] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM symposium on principles of distributed computing*. 347–356.
- [113] Dachao Yu, Hao Xu, Lei Zhang, Bin Cao, and Muhammad Ali Imran. 2021. Security analysis of sharding in the blockchain system. In *2021 IEEE 32nd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 1030–1035.
- [114] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 511–526.
- [115] Jianting Zhang, Zhongtang Luo, Raghavendra Ramesh, and Aniket Kate. 2024. Optimal Sharding for Scalable Blockchains with Deconstructed SMR. *arXiv preprint arXiv:2406.08252* (2024).
- [116] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient distributed transaction processing in heterogeneous networks. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1372–1385.
- [117] Shijie Zhang, Ru Cheng, Xinpeng Liu, Jiang Xiao, Hai Jin, and Bo Li. 2024. Seer: Accelerating Blockchain Transaction Execution by Fine-Grained Branch Prediction. *Proceedings of the VLDB Endowment* 18, 3 (2024), 822–835.
- [118] Shijie Zhang, Jiang Xiao, Enping Wu, Feng Cheng, Bo Li, Wei Wang, and Hai Jin. 2024. Morphdag: A workload-aware elastic dag-based blockchain. *IEEE Transactions on Knowledge and Data Engineering* 36, 10 (2024), 5249–5264.
- [119] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent state machines for recoverable in-memory storage systems with {NVRam}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1029–1046.
- [120] Yunhao Zhang, Haobin Ni, Soumya Basu, Shir Cohen, Maofan Yin, Lorenzo Alvisi, Robbert van Renesse, Qi Chen, and Lidong Zhou. 2025. Ordered Consensus with Equal Opportunity. *arXiv:2509.09868 [cs.DC]* <https://arxiv.org/abs/2509.09868>
- [121] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 633–649.
- [122] Jianjun Zhao, Haikun Liu, Shuhao Zhang, Haodi Lu, Yancan Mao, Zhuohui Duan, Xiaofei Liao, and Hai Jin. 2025. Towards {High-Performance} Transactional Stateful Serverless Workflows with {Affinity-Aware} Leasing. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1535–1551.
- [123] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2939–2952.

APPENDIX

A SYSTEM SUMMARY

A.1 Entities

A validator in REMORA consists of two types of entities that work together to execute transactions. The **Coordinator** serves as the validator’s centralized coordinator, performing the following functions: it interfaces with the consensus layer, assigns per-key versions in input order, schedules and dispatches transactions to workers, tracks object ownership through leases, controls elasticity by managing the worker pool, persists snapshots for durability, and coordinates recovery after failures.

The **Workers** are a dynamic set of execution nodes responsible for transaction processing. These nodes execute both stateless and stateful parts of transactions, maintain leased objects in memory, fetch objects and accept lease transfers on demand from other workers, and perform local readiness gating to determine when transactions can execute. Unlike the Coordinator, workers do not engage in global scheduling decisions, focusing instead on local execution.

Additionally, a Consensus module runs on the Coordinator, producing committed blocks that establish a total order of transactions. A *block* is an ordered batch of transactions that are processed atomically by the consensus layer. The Coordinator can also observe blocks proposed to consensus before their commitment, enabling speculative execution.

A.2 State and Objects

The global state in REMORA is represented as a versioned map $(\mathcal{K} \times \mathbb{N}) \rightarrow \mathcal{V}$, where each pair (k, i) identifies the value of object k at version index i .

For each key k , the Coordinator maintains a monotonically increasing counter $\text{ver}[k] \in \mathbb{N}$ that tracks the next version index to be assigned. An *object version* is defined as a pair (k, i) where $0 \leq i \leq \text{ver}[k]$. The system assumes an initial state where every key k has an initial version $(k, 0)$ representing its initial value.

A.3 Transactions and Separation

Each transaction T declares static read/write sets $R(T), W(T) \subseteq \mathcal{K}$ that are known to the Coordinator before scheduling begins. This static declaration enables the Coordinator to perform dependency analysis and version assignment without executing the transaction.

Transaction execution follows a two-phase separation model. The **stateless step** consists of a validation function $\text{Validate}(T)$ that returns either a token τ_T or an invalid result. This token is bound to the transaction’s inputs, ensuring consistency between validation and execution. The **stateful step** takes the validation token τ_T along with the object versions assigned to T by the Coordinator, then performs state reads and writes before terminating.

When $\text{Validate}(T)$ returns invalid, the stateful step still executes but performs a no-op operation. This no-op consumes the assigned versions to preserve the per-key version streams, ensuring that version numbers remain consistent across all transactions.

A.4 Time and Consensus Interface

The system distinguishes between speculative and committed work through its consensus interface. Pre-consensus work on proposals is speculative and maintains the invariant of being side-effect free on state. The system uses consensus rounds to define logical epochs $E \in \mathbb{N}$, which serve as boundaries for lease expiration and provide a synchronization point for system-wide operations.

A.5 Communication

Communication between Workers is controlled to maintain system integrity. Workers may engage in peer-to-peer communication solely for fetching object state when authorized by the Coordinator through a lease transfer. This restriction ensures that all state movement is tracked and authorized centrally while allowing direct transfers between execution nodes.

A.6 Version Assignment (Coordinator)

The Coordinator assigns versions to transactions deterministically based on their position in the consensus order. For a sequence of transactions $\langle T_1, \dots, T_n \rangle$, the Coordinator performs version assignment by walking through the order exactly once. For each transaction T_i and for each key $k \in R(T_i) \cup W(T_i)$, the Coordinator increments the version counter $\text{ver}[k] \leftarrow \text{ver}[k] + 1$ and assigns the input version $(k, \text{ver}[k])$ to transaction T_i . This mechanism ensures the property of *single-consumption*: each object version (k, i) is consumed by exactly one transaction, after which it becomes eligible for garbage collection.

To enable speculative execution during the consensus window, the Coordinator applies the same version assignment rule to proposed transaction orders using separate speculative counters $\text{ver}^{\text{spec}}[\cdot]$. If the committed order matches the proposal, the speculative assignment is adopted, allowing transactions to proceed without recomputation. Otherwise, the speculative assignment is discarded and the Coordinator recomputes versions based on the committed order.

A.7 Ownership and Leasing

The system maintains an ownership model where each key k has at most one owner Worker at any given time. The Coordinator serves as the authoritative source for ownership metadata, maintaining $\text{owner}[k]$ for every key in the system to ensure consistency and prevent conflicting ownership claims.

Lease transfers enable state movement between workers when transaction placement requires it. When a transaction T is placed on Worker A but requires a key k currently owned by Worker B , the Coordinator orchestrates the transfer by including a source hint B in the dispatch message to A and atomically updating the ownership metadata $\text{owner}[k] \leftarrow A$. Worker A then fetches the required version (k, i') where $i' \leq i$ from Worker B through peer-to-peer communication. If Worker B has already checkpointed its state or is unavailable, Worker A fetches the data from the Coordinator.

At epoch boundaries, the system performs ownership reconciliation. Every worker takes a snapshot of all objects it owns that were modified during epoch E and reports this information to the Coordinator. This reporting ensures that the Coordinator maintains

a view of the global state distribution for garbage collection and recovery operations.

A.8 Persistence and Recovery

At each epoch E , the Coordinator constructs a global snapshot by combining lease-aligned reports from all Workers. These reports contain the latest versions of objects owned by each worker that were modified during the epoch. After the Coordinator persists the snapshot and acknowledges receipt, workers may garbage collect versions with indices less than or equal to E according to the configured policy.

When a Worker crashes, the Coordinator selects the latest snapshot epoch E , spawns replacement workers, and seeds them with state from the snapshot. The replacement workers then replay the committed transaction log from round $E+1$ to the current round, reconstructing the lost state through deterministic execution.

Garbage collection occurs at two points. During normal execution, when a worker creates a new version of an object, it may garbage collect all older versions of that object. At epoch boundaries, after sending its report to the Coordinator and receiving acknowledgment, a worker may garbage collect all but the latest version of each object it owns.

A.9 Execution Rule at Workers

A Worker executes a stateful transaction T only when all Coordinator-assigned input versions (k, i) for $k \in R(T) \cup W(T)$ are available locally, either produced by prior local execution or fetched through lease transfer. This availability requirement ensures that transactions execute with the input versions determined by the global ordering.

When reading version (k, i) , the worker returns the value produced by the unique transaction to access k preceding T in the committed order. If no such transaction exists, the read returns the value from the snapshot or the initial value. Executing T consumes all assigned input versions (k, i) and produces new versions $(k, i+1)$ for each $k \in W(T)$ with the updated values.

Before consensus commitment, Workers may execute only stateless validation. No state reads or writes, lease transfers, or version consumption occur during the pre-consensus phase, maintaining the invariant that speculative work has no side effects on persistent state.

A.10 Notation Summary

- $\mathcal{K}$: The set of all keys in the system
- $\mathcal{V}$: The domain of values that objects can hold
- $R(T)$: The declared read set of transaction T
- $W(T)$: The declared write set of transaction T
- $\text{ver}[k]$: The next version index for key k maintained at the Coordinator
- (k, i) : An object version, pairing key k with version index i
- $\text{owner}[k]$: The current owner Worker of key k
- E : An epoch corresponding to a consensus round
- τ_T : The validation token for transaction T

B ASSUMPTIONS

REMORA makes the following assumptions to ensure correctness:

Core System Assumptions.

- **A1 (Consensus)**: The consensus layer provides a total order over transactions in committed blocks with safety (no forks) and eventual liveness.
- **A2 (Determinism)**: Stateless and stateful execution are deterministic given their inputs, with no hidden sources of nondeterminism (e.g., system time).
- **A3 (Cryptography)**: Tokens produced by $\text{Validate}(T)$ are unforgeable, bound to T 's inputs, and verifiable by the stateful step.

Communication and Timing.

- **A4 (Eventual Delivery)**: Messages are eventually delivered despite possible delay, reordering, or duplication; retry ensures eventual delivery.
- **A5 (Local Liveness)**: If a worker starts executing the stateless or stateful part of a transaction and does not crash, it eventually completes execution.
- **A6 (Partial Synchrony)**: The system is partially synchronous with an eventually perfect failure detector: crashed workers are eventually suspected (completeness) and correct workers are eventually not suspected (accuracy).

Failure Model.

- **A7 (Worker Failures)**: Workers are crash-stop and may be replaced. No Byzantine behavior occurs within a validator.
- **A8 (Stability)**: Eventually the inter-crash interval is sufficiently long to allow recovery to complete.
- **A9 (Router Durability)**: The Coordinator remains available and its storage persists across worker failures. The Coordinator is a single point of failure.

Performance Optimizations. The following assumption enables speculation but is not required for correctness:

- **Proposal Visibility**: The Coordinator may observe proposed blocks and their intra-block order before commitment.

C PSEUDOCODE

Algorithms 2–5 show the main functions of the Coordinator and Workers. To simplify the presentation, we assume the existence of some black-box helper functions. These are typeset in monospace. To distinguish the functions we define from the black-box helpers, we typeset the former in SMALL CAPS.

Helpers (black boxes).

- **AssignVersions**: Assign per-key version numbers to each transaction in block order.
- **BuildDependencies**: Construct the intra-block dependency graph from read/write version relationships.
- **Partition**: Group the dependency graph into execution units that preserve dependencies.
- **PlaceUnits**: Map execution units to workers according to placement policy (load/locality/balance).

Algorithm 2: Coordinator main functions

```
1 Procedure ONPROPOSEDBLOCK(block)
2   RUNSTATELESS(block)
3   specPlan ← PLANBLOCK(block)
4 Procedure ONCOMMITTEDBLOCK(block)
5   if MatchesProposal(block) then
6     plan ← specPlan
7   else
8     plan ← PLANBLOCK(block)
9   DISPATCH(plan)
10 Procedure PLANBLOCK(block)
11   vers ← AssignVersions(block)
12   deps ← BuildDependencies(block)
13   units ← Partition(deps)
14   place ← PlaceUnits(units)
15   return {vers, place}
16 Procedure RUNSTATELESS(block)
17   foreach txn ∈ block do
18     ScheduleStateless(txn)
19 Procedure DISPATCH(plan)
20   foreach txn ∈ plan.block do
21     vers ← PlanVers(txn)
22     place ← PlanPlace(txn)
23     sources ← PlanSources(txn)
24     foreach (k, i) ∈ vers do
25       SetOwner(k, place)
26     SendToWorker(txn, vers, place, sources)
```

- ScheduleStateless: Enqueue stateless validation tasks for execution (pre-commit, no state).
- Validate: Run the stateless function; return a token or invalid.
- PublishToken: Make a transaction's validation token available to consumers.
- AwaitToken: Block until the transaction's token is available (or invalid).
- AwaitAllAvailable: Ensure all required versions are locally available by driving peer fetches using source hints and falling back to the Coordinator.
- ConsumeNoOp: Consume assigned versions without changing state when validation is invalid.
- Read: Materialize values for the assigned read-set versions from local storage.
- Execute: Run the stateful logic using the read set and token to compute new values.
- Write: Persist new values to the assigned write-set versions and make them available.
- Ack: Acknowledge transaction completion to the coordinator.
- Owner: Return the current worker that holds the lease for a key.
- SetOwner: Update the authoritative owner of a key after transfer.

Algorithm 3: Worker main functions

```
1 Procedure ONSTATELESS (txn)
2   token ← Validate(txn)
3 Procedure ONSTATEFUL (txn, versions, sources)
4   ENSUREOWNERSHIP(versions, sources)
5   AwaitAllAvailable(versions, sources)
6   token ← AwaitToken(txn)
7   if Invalid(token) then
8     ConsumeNoOp(versions)
9     return
10  readSet ← Read(versions.read)
11  newVals ← Execute(txn, readSet, token)
12  Write(versions.write, newVals)
13 Procedure ENSUREOWNERSHIP (versions, sources)
14   foreach (k, i) ∈ versions do
15     while not HasVersion(self, k, i) do
16       hint ← sources[k]
17       verReq ← LatestLeq(i)
18       ok ← PeerFetch(self, hint, k, verReq)
19       if not ok then
20         FetchFromRouter(self, k, verReq)
```

Algorithm 4: Persistence and Recovery (on Coordinator)

```
1 Procedure ONEPOCHTRIGGER (E)
2   RequestSnapshot(E)
3   CollectAndPersist(E, reports)
4 Procedure ONWORKERFAILURE (failed)
5   E ← LatestSnapshotEpoch()
6   new ← SpawnWorkers()
7   Seed(new, snapshot[E])
8   range ← LogRange(E+1, head)
9   Replay(new, range)
10  PromoteReplacement(new)
```

Algorithm 5: Elasticity (on Coordinator)

```
1 Procedure SCALEOUT ()
2   w ← SpawnWorker()
3   BiasStateless(w)
4   GraduallyPlaceStateful(w)
5 Procedure SCALEIN (worker)
6   StopPlacingStateful(worker)
7   BiasStateless(worker)
8   WaitForLeasesToTransferOrExpire()
9   Shutdown(worker)
```

- HasVersion: Check if a worker holds a specific key version locally.
- AuthorizeTransfer: Approve a lease transfer from current to target owner. (Coordinator-side metadata only.)

- **PeerFetch**: Worker-to-Worker transfer of the requested key state ($\leq$ requested version) using the Coordinator-provided source hint.
- **FetchFromCoordinator**: Fallback transfer of the requested key state ($\leq$ requested version) from the Coordinator when the current owner has checkpointed or is unavailable.
- **RequestSnapshot**: Instruct workers to snapshot at epoch E and report. Ack receipt so that workers can garbage collect.
- **CollectAndPersist**: Aggregate snapshot reports and persist a global snapshot.
- **LatestSnapshotEpoch**: Return the latest fully persisted snapshot epoch.
- **SpawnWorkers**: Start replacement worker processes for recovery.
- **Seed**: Load snapshot state into newly spawned workers.
- **Replay**: Deterministically re-execute committed log entries since the snapshot.
- **PromoteReplacement**: Promote a single recovered worker to replace the failed one and shut down the rest.
- **Shutdown**: Cleanly stop a worker after it is drained.
- **BiasStateless**: Prefer routing stateless tasks to the specified worker.
- **GraduallyPlaceStateful**: Incrementally assign stateful work/leases to the worker.
- **StopPlacingStateful**: Stop assigning new stateful work to the worker.
- **WaitForLeasesToTransferOrExpire**: Wait until the worker's leases move or expire at epochs.

D CORRECTNESS PROPERTIES

REMORA satisfies the following correctness properties:

THEOREM 1 (CONSENSUS-ORDERED SERIALIZABILITY). *For a correct validator, execution under REMORA of the transactions $\{T_1, \dots, T_n\}$ yields the same state as if the transactions were executed sequentially, in consensus order.*

THEOREM 2 (CROSS-VALIDATOR DETERMINISM). *No two correct validators that execute the same sequence of transactions $\langle T_1, \dots, T_n \rangle$ yield different states.*

THEOREM 3 (LIVENESS). *For every committed T_i , REMORA eventually executes T_i and incorporates its effect into the global state.*

E PROOFS

E.1 Serializability

We prove Theorem 1. Let $\langle T_1, \dots, T_n \rangle$ be the total transaction order produced by consensus. Let $R(T)$ and $W(T)$ denote the read/write sets of transaction T (§A). The Coordinator assigns versions according to §A.6 and Workers execute according to §A.9. We begin with the following helpful invariants.

LEMMA 4 (PER-KEY VERSIONS ARE CONSECUTIVE). *For every key $k \in \mathcal{K}$, the sequence of versions $(k, 0), (k, 1), \dots$ is a linear chain with no gaps (i.e., the versions are consecutive), and each version is*

assigned at most once in the consensus order of the transactions that touch k .

PROOF. This follows directly from the Coordinator's assignment rule: for each T_i and each $k \in R(T_i) \cup W(T_i)$, the Coordinator increments $\text{ver}[k]$ once and assigns the resulting $(k, \text{ver}[k])$ to T_i (§A.6), in consensus order. $\square$

LEMMA 5 (READ FROM LATEST ACCESS). *When a transaction T reads key k , it observes the value produced by the latest transaction that precedes T in the consensus order and accesses k , or the initial value if no such transaction exists.*

PROOF. Let T' be the latest transaction that precedes T in the consensus order and accesses k , or the initial state if no such transaction exists. By the version assignment rule (§A.6), versions are assigned consecutively in consensus order. Thus, T reads the value produced by T' . $\square$

LEMMA 6 (NO PRE-COMMIT STATE EFFECTS). *Pre-consensus actions do not read or mutate state, nor consume versions or transfer leases, hence cannot affect post-commit execution results.*

PROOF. By §A.9, Workers only perform stateless validation before consensus (Algorithm 2, Line 18); they do not perform any state reads/writes, transfer any leases, or consume any versions before commitment. $\square$

LEMMA 7 (SPECULATIVE ASSIGNMENT IS SAFE). *The version assignment of the Coordinator equals that induced by the committed order.*

PROOF. If a proposed block's order matches the committed order, adopting the speculative version plan is equivalent to recomputing on the committed order; otherwise, speculation is discarded and the committed-order plan is used. In either case, the effective version assignment equals that induced by the committed order. $\square$

LEMMA 8 (VERSION VALUE UNIQUENESS). *For any key k and version index i , if version (k, i) exists at multiple locations in the system (e.g., at different workers, in persistent storage, or produced by re-execution during recovery), all copies have identical values.*

PROOF. By contradiction. Suppose there exist two copies of version (k, i) with different values v and v' . Let (k, i) be the earliest such version (in consensus order of producing transactions) where a disagreement occurs.

By the single-consumption property (Lemma 4), exactly one transaction T in the consensus order produces version (k, i) by consuming $(k, i - 1)$ and writing to k .

Transaction T may be executed multiple times (e.g., once before a failure and once during recovery). In each execution, T reads the same input versions: for each $k' \in R(T)$, it reads version $(k', j_{k'})$ where $j_{k'}$ is determined by the version assignment. By minimality of (k, i) , all versions earlier than (k, i) in consensus order have unique values, so T observes identical input values in each execution.

Since T has the same validation token τ_T (determined by consensus) and reads identical input values, by the determinism assumption (§B), T must produce the same output value for key k . Therefore, both copies of (k, i) must have the same value. Contradiction. $\square$

Constructing the sequential history. Let H be the execution history in which transactions are applied sequentially in consensus order. We show by induction on i that the state after completing T_i in REMORA equals the state after applying $T_1, \dots, T_i$ sequentially in H , assuming the same validation outcomes for $T_1, \dots, T_i$ in both cases. Note that by Lemma 8, even if failures and recovery cause transactions to be re-executed, each version has a unique, well-defined value throughout the system, ensuring that the state remains consistent despite potential re-executions.

LEMMA 9 (PREFIX EQUIVALENCE). *For all $i \in \{0, \dots, n\}$, after REMORA has executed $T_1, \dots, T_i$, the value of every key version (k, j) equals that in the sequential execution of $T_1, \dots, T_i$.*

PROOF. By induction on i . Base $i = 0$ holds because the initial state is the same. For the inductive step, fix $i > 0$ and assume the claim for $i - 1$. Consider T_i :

- If $\text{Validate}(T_i)$ is invalid, REMORA executes a no-op that consumes assigned versions (§A.3) so the state remains unchanged; the sequential execution also skips writes. By Lemma 7, the version assignment produced by the Coordinator equals the one induced by consensus. Thus, the state produced by REMORA coincides with the state produced by the sequential execution after T_i .
- If $\text{Validate}(T_i)$ is valid, by Lemma 5, the read set observed by T_i in REMORA equals the values accessed by the latest predecessors in $\langle T_1, \dots, T_{i-1} \rangle$ (or initial values), which by the inductive hypothesis equal those in the sequential prefix state. By the determinism assumption (§B) and Lemma 7, the stateful computation produces the same outputs as the sequential execution would on the same inputs. REMORA then writes them to $(k, j + 1)$ for $k \in W(T_i)$ and consumes (k, j) as specified, matching the sequential step. Keys not touched by T_i are unchanged.

Thus, the state produced by REMORA coincides with the state produced by the sequential execution after T_i , completing the inductive step. $\square$

PROOF OF THEOREM 1. By Lemma 9 with $i = n$, the state after completing all transactions in REMORA equals the state after sequential execution in consensus order. Therefore, REMORA is serializable with respect to the consensus order. $\square$

E.2 Determinism

We derive Theorem 2 as a corollary of serializability and the determinism assumption (§B).

PROOF OF THEOREM 2. Let V and V' be two correct validators that execute the same committed sequence $\langle T_1, \dots, T_n \rangle$ from the same initial snapshot. By Theorem 1, each validator's execution is equivalent to executing $\langle T_1, \dots, T_n \rangle$ sequentially in consensus order. By the determinism assumption, stateless validation outcomes and stateful effects are deterministic functions of their inputs and reads; therefore both sequential executions produce identical per-transaction effects (invalid transactions perform consuming no-ops) and thus identical final states. Hence V and V' cannot diverge. $\square$

E.3 Liveness

We prove Theorem 3. Intuitively, we first show that REMORA's scheduler is fair and then, given fairness at the Coordinator, eventual delivery in the network, and partial synchrony, we show that every committed transaction is eventually dispatched, validated, and its stateful step executed, despite worker failures.

E.3.1 Fairness.

THEOREM 10 (SCHEDULER FAIRNESS). *Fix any finite committed block B . After $\text{ONCOMMITTEDBLOCK}(B)$ the Coordinator eventually dispatches every transaction $T \in B$ to some Worker.*

PROOF. Upon commitment of B , the Coordinator either adopts the precomputed stateful plan (if the proposal order matches) or recomputes it on the committed order. In either case, planning proceeds as follows: it builds the dependency graph $D(B)$ and extracts its disconnected components $\mathcal{G}$, which form a partition of the finite set B . It then iterates once over each $G \in \mathcal{G}$, computes a score S_i for every worker i from a finite worker set, and selects $i^* = \arg \max_i S_i$ (ties broken arbitrarily), thereby assigning G —and thus every $T \in G$ —to i^* . Because both $\mathcal{G}$ and the worker set are finite, this placement completes in finite time and yields a target worker for every $T \in B$.

The Coordinator then executes dispatch over the finite set B , invoking SendToWorker once per T . By eventual delivery, each dispatch message reaches its assigned Worker. Therefore, every committed transaction $T \in B$ is eventually dispatched, establishing scheduler fairness. $\square$

E.3.2 Liveness Given Fairness.

REMARK 11 (NO CIRCULAR DEPENDENCIES). *Transactions cannot deadlock due to circular version dependencies.*

PROOF. The version assignment rule ensures that transaction T_i at position i in consensus order $\langle T_1, \dots, T_n \rangle$ can only depend on versions produced by transactions T_j where $j < i$. This creates a directed acyclic graph where dependency edges point strictly backward in consensus order. Since the dependency relation respects the total order of consensus, cycles are impossible. $\square$

PROOF OF THEOREM 3. We consider the failure-free case first and later extend to the general case. Let $\langle T_1, T_2, \dots \rangle$ be the transaction order output by consensus. We prove by induction on $i \geq 1$ that T_i eventually executes.

Base $i = 1$. By scheduler fairness and eventual delivery (§B), the Coordinator eventually dispatches the stateless and stateful steps of T_1 to some Worker or Workers. By local liveness, the stateless step of T_1 eventually completes and its validation token becomes available to the stateful step. For each required input key $k \in R(T_1) \cup W(T_1)$, the corresponding version is the initial version $(k, 0)$, which is available at the Coordinator's persisted state. The Worker requests these versions if necessary and eventually acquires them (Algorithm 3, Line 4–Line 5), as well as the token (Algorithm 3, Line 6). With the token and input versions available and by local liveness, the Worker eventually executes the stateful step (or the consuming no-op, if invalid), completing T_1 .

Inductive step. Assume $T_1, \dots, T_{i-1}$ eventually execute, and consider T_i . By fairness and eventual delivery, T_i is dispatched to some

Worker. The stateless step of T_i is scheduled and, by local liveness, its validation token is eventually computed (Algorithm 3, Line 2) and obtained by the Worker assigned the stateful step (Algorithm 3, Line 6). For each key k that T_i requires as input, let T' be the latest transaction to access k preceding T_i in consensus order (if none such transaction exists, interpret T' as the creation of the initial state). By the induction hypothesis, T' has completed (note that T' precedes T_i , so no circular dependency exists by Remark 11), thus the required version for k exists at its owner (or at the Coordinator's persisted state). The Worker requests this version if necessary and eventually acquires it (Algorithm 3, Line 4–Line 5). With the token and inputs available, by local liveness, the Worker eventually performs the stateful step (or the consuming no-op if invalid), completing T_i . This completes the induction.

We now allow Worker failures (crash-stop) and argue that failures do not prevent the eventual completion of any committed T_i . Assume that a worker W executing the stateless or stateful step of T_i crashes. The Coordinator detects the failure and initiates recovery (Algorithm 4, Line 4). It spawns replacements and seeds them from the latest persisted snapshot (Algorithm 4, Line 7), which cannot include T_i , since T_i has not completed yet. The Coordinator then deterministically replays the committed log up to the current head (Algorithm 4, Line 10), after which it promotes the replacement worker W' and shuts down the rest. It is possible that Worker crashes interrupt the recovery process, but by the stability assumption (§B), there will eventually be a long enough failure-free interval to complete the recovery and finish executing T_i . $\square$