

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Ira: Efficient Transaction Replay for Distributed Systems

Adithya Bhat, Harshal Bhadreshkumar Shah, and Mohsen Minaei, *Visa Research*

https://www.usenix.org/conference/usenixsecurity26/presentation/bhat

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

Ira: Efficient Transaction Replay for Distributed Systems

Adithya Bhat
Visa Research

Harshal Bhadreshkumar Shah
Visa Research

Mohsen Minaei
Visa Research

Abstract

In primary-backup replication, the consensus latency is bounded by the time for backup nodes to replay (re-execute) transactions proposed by the primary. Our key insight is that the primary, having already executed transactions, possesses knowledge of future access patterns which is the information needed for optimal replay by the backups. In this work, we present Ira, a framework to accelerate backup replay by transmitting compact *hints* alongside transaction batches.

We use Ethereum for our case study and present a concrete protocol, Ira-L, within our framework to improve cache management of Ethereum block execution. The primaries implementing Ira-L provide hints that consist of the working set of keys used in an Ethereum block and one byte of metadata per key indicating the table to read from, and backups use these hints for efficient block replay.

We evaluated Ira-L against the state-of-the-art Ethereum client reth over two weeks of Ethereum blocks (100,800 blocks, 24 million transactions). Our hint generation adds 10.9% overhead to primary execution time. On the backup, our hint-driven prefetching speeds up aggregate replay by 5.2× with a single prefetch thread, and by 23.6× with 16 threads. Our hints add a median of 47 KB compressed metadata per block (∼5% of block payload).

1 Introduction

The primary-backup paradigm is the cornerstone of fault-tolerant distributed systems [36], from traditional databases [40, 44] to modern blockchain networks [71, 77]. In the standard State Machine Replication (SMR) model [62], a designated *primary* executes batches of transactions and propagates them to *backup* replicas, who must replay these transactions to maintain consistent state.

We model each transaction as a sequence of read and write operations against a key-value store. The primary executes these operations, leveraging caches and indexes to optimize performance. When propagating the transaction batch to backups, the current systems transmit only the transaction data itself, forcing each backup to independently reconstruct the execution path. Transmitting all read/write values would incur prohibitive bandwidth, whereas replaying compact transaction logs allows backups to independently reconstruct the state.

SMR protocols [48, 58] run a consensus protocol to ensure consistency of the state machine, but the latency of consensus depends on the response from a quorum of replicas (typically a majority or a super-majority). This makes replay critical to consensus performance.

This replay is inherently inefficient as the backups must make decisions (e.g., cache management) without knowledge of future access patterns, leading to sub-optimal eviction choices such as unnecessary waiting for I/O, redundant disk I/O, etc.

We illustrate this in Fig. 1. Consider a cache of size 2 containing keys $\{A, B\}$, with upcoming accesses C, A, C. When C arrives, we must evict one key. LRU evicts the least recently used key (A), causing a miss when we access A next. An optimal strategy would be to evict B avoiding the subsequent miss on A. LRU incurs 2 misses while optimal incurs only 1. Generalizing this, without knowledge of future accesses, backups using LRU or other heuristic policies cannot make these optimal decisions.

1.1 Our Approach: Hint-Based Replay

When the cache is smaller than the working set, Belady's MIN algorithm [21, 52] provides the optimal eviction policy: evict the key whose next access is furthest in the future. However, when the cache can hold the entire working set, a simpler and more effective strategy is *prefetching*: load all required keys before execution begins, eliminating misses entirely. In primary-backup replication, the primary already knows the complete access set—this knowledge enables prefetching on backups.

Our key insight is that the inefficiency from Fig. 1 stems from *information asymmetry*: the primary possesses complete knowledge of the access pattern but discards this informa-

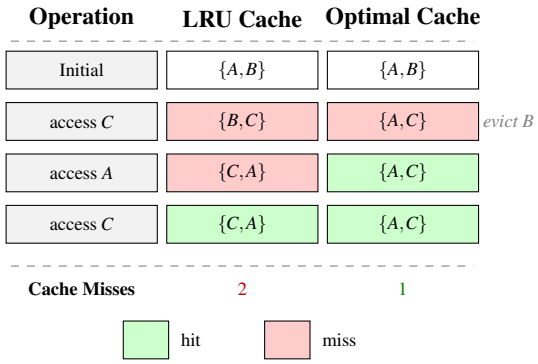


Figure 1: LRU vs. optimal caching (cache size is 2). For access sequence C, A, C starting with cache $\{A,B\}$: LRU evicts A (least recently used) when C arrives, then misses when it accesses A next. Optimal evicts B (never accessed again), achieving one fewer miss. With hints revealing the access set, backups can prefetch all required keys before execution, eliminating misses entirely.

tion when communicating with backups. The primary can encode this knowledge compactly and transmit it as *hints* that enable backups to implement near-optimal replay. We propose Ira, a framework to accelerate replay through hint-based replay. Figure 2 illustrates the architecture. Ira operates two components:

Primary. This component executes transactions while recording keys and their accesses, producing a hint. The concrete details of the hints depend on the execution and storage details of the application. This component is also responsible for delivering the hint to the backups.

Backups. This component replays transactions using the hint to improve the storage access. This component is also responsible for receiving the hints from the primary.

1.2 Contributions

In summary, we make the following contributions:

- **Ethereum Case Study:** We systematically perform measurement studies on execution in Ethereum. We identify that I/O is a bottleneck, and caching plays a major role in improving the replay performance.
- **Hint-Based Replay Architecture:** We design a system where the primary generates compact hints during execution that enable backups to implement optimal caching. Hints are advisory, i.e., incorrect hints affect performance but do not affect correctness, thus enabling incremental deployment.
- **Ira for Ethereum.** We present Ira-L, which achieves zero-miss cache performance through per-block hints that enable complete working-set prefetching.
- **Performance evaluation:** We implement Ira by modifying the state-of-the-art Rust Ethereum implementation (reth) and demonstrate 24.9× faster replay with only 10.9% hint generation overhead. With even a single backup node, Ira achieves net speedup; benefits grow with additional replicas.

2 Model and Preliminaries

Cache Optimization with Future Knowledge. Belady's MIN algorithm [21], proved optimal by Mattson et al. [52], provides the theoretically optimal cache replacement policy: when eviction is necessary, evict the item whose next access is *furthest in the future*. However, when the cache can hold the entire working set, a simpler strategy—*prefetching*—is more effective: load all required keys before execution begins, eliminating misses entirely. Both strategies require knowledge of future accesses, information that is typically unavailable but is available to the primary in primary-backup replication (Section 2.3).

2.1 Primary-backup system model

We assume the standard primary-backup replication model.

Primary Node. The primary executes transactions against a key-value store (e.g., MDBX) and proposes a batch of transactions for replication.

Backup Nodes. Backups receive transaction batches and associated metadata from the primary. The backups replay the transactions with standard configured cache policies (e.g., LRU). Backups independently verify execution results, ensuring correctness of primary's proposal.

Execution Flow. A transaction batch proceeds through the system as follows:

1. The primary executes the batch, generating state updates.
2. The primary commits the batch to durable storage and transmits the log to backups.
3. Each backup replays the batch.
4. Backups acknowledge replay completion to the primary.

2.2 Ethereum: A Primer

Ethereum [71] is a decentralized blockchain platform that executes arbitrary programs called *smart contracts*. The execution client runs the Ethereum Virtual Machine (EVM), a stack-based virtual machine that interprets contract bytecode. We provide background on the aspects relevant to understanding Ira-L.

Client Architecture. Ethereum nodes run two cooperating clients: a *consensus client* that participates in the proof-of-stake protocol to determine the canonical chain, and an *execution client* that executes transactions and maintains state.

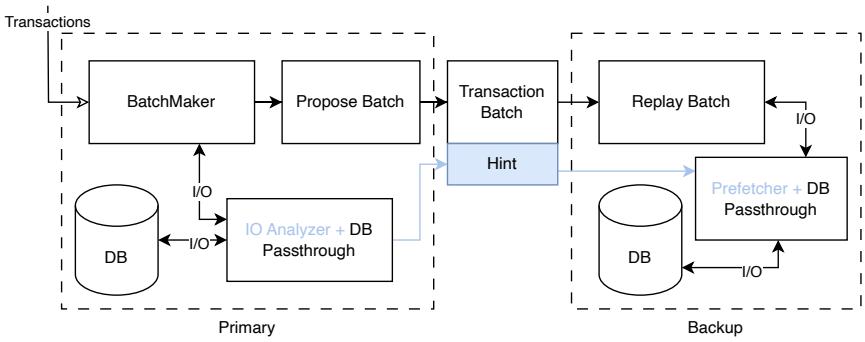


Figure 2: Ira architecture. The primary executes transactions while building a *hint* that records the access set. Backups use these hints to prefetch all required state before replay, achieving zero cache misses.

The consensus client receives blocks from the network and instructs the execution client to validate them by re-executing all transactions and verifying the resulting *state root*, the cryptographic commitment to all account state.

Blocks and Transactions. A block contains an ordered sequence of transactions. Each transaction specifies a sender, recipient, value transfer, and optional calldata for smart contract invocation. The execution client processes transactions sequentially: each transaction observes the state modifications of all preceding transactions in the same block. This deterministic execution model ensures that all nodes converge to identical state.

State Model. Ethereum state is a mapping from 20-byte *addresses* to *accounts*. Each account contains:

Balance. The account's Ether holdings (256-bit integer)

Nonce. A counter incremented with each outgoing transaction

Code. For contract accounts, the immutable bytecode (up to 24 KB)

Storage. For contract accounts, a mapping from 32-byte slots to 32-byte values

Contract storage is the dominant source of state I/O: a single contract may have millions of storage slots, and complex DeFi protocols access hundreds of slots per transaction.

The EVM is a stack-based virtual machine that executes contract bytecode. State-accessing opcodes include:

`SLOAD`/`SSTORE`: Read/write contract storage slots

`BALANCE`/`SELFBALANCE`: Query account balances

`CALL`/`STATICCALL`/`DELEGATECALL`: Invoke other contracts (loads bytecode)

`EXTCODESIZE`/`EXTCODECOPY`: Query contract bytecode

Each state-accessing opcode translates to one or more database lookups in the execution client. Section 3 characterizes the distribution and locality of these operations.

Block Validation as Primary-Backup Replay. In Ethereum's proof-of-stake consensus, a designated *proposer* (analogous to the primary) constructs and executes a new block, then broadcasts it to *validators* (analogous to backups). Validators must re-execute the block to verify correctness before attesting to it. This replay is the bottleneck that Ira-L addresses.

2.3 Ira: Hint-Based Replay Architecture

Ira addresses the information asymmetry between primary and backups by having the primary encode its access pattern as a compact *hint* and transmit it alongside the transaction batch. The hint contains sufficient information for backups to prefetch required state before replay, eliminating cache misses. Critically, hints are *advisory*: they affect only performance, not correctness. If a hint is missing, corrupted, or incomplete, the backup falls back to standard replay without prefetching.

Protocol Overview. The Ira protocol operates in five phases:

1. **Primary Execution:** The primary executes transactions while instrumenting key accesses, recording each key read or written into an *access set A*.
2. **Hint Construction:** After execution, the primary constructs a hint H containing the access set A, optionally annotated with metadata (e.g., where to read each key, or access ordering if cache capacity is limited).
3. **Hint Transmission:** The primary transmits H alongside (or ahead of) the transaction batch to backups.
4. **Backup Prefetch:** Backups use H to prefetch all keys in A into their cache before execution begins.
5. **Backup Replay:** Backups execute transactions with all required state pre-cached, achieving near-zero I/O latency during execution.

Hint Contents. At minimum, a hint contains the *access set A*: the set of unique keys accessed during batch execution. Richer hints may include additional metadata:

- **Source annotations:** Where to read each key (e.g., the *main state table* that holds each key's current value vs.

historical change sets that record prior values), enabling the backup to route reads to the fastest storage path.
- **Access ordering:** The sequence in which keys are accessed, enabling optimal cache eviction when the access set exceeds cache capacity.
- **Read values:** The actual values read by the primary, trading bandwidth for backup computation when network capacity exceeds storage throughput.

The choice of hint granularity depends on the system's I/O characteristics, network bandwidth, and memory constraints. Section 4 presents Ira-L, a concrete instantiation that uses per-block access sets with source annotations for Ethereum block replay. We defer the discussion of how to adapt Ira to other primary-backup systems (Raft-based databases, streaming replication, distributed key-value stores) to the full version [24].

Threat Model and Security Goals. Ira is a performance optimization assuming that the underlying replication protocol provides consensus ensuring safety and liveness. Our system Ira inherits these guarantees. Our hints are *advisory* and affect only replay latency.

Trust. We assume (i) the replication protocol is secure, i.e., ensures safety and liveness, and (ii) backups verify execution outputs against the primary's commitment, e.g., the Ethereum state root (Section 4.3). The primary may be Byzantine.

Adversary. The adversary controls the primary and any nodes the underlying protocol tolerates such as the clients, and schedules hint and block delivery within the protocol's network assumptions. Against Ira specifically, it may (a) omit hints, (b) send a hint that fails an integrity check, (c) send a valid hint that omits required keys, or (d) send a hint with spurious keys. Our security goal is that none of these cause a backup to accept an incorrect result or to diverge from the state it would compute without hints; a malicious hint may affect only replay latency. Section 4.3 and Section D establish how Ira-L detects and absorbs each case.

Security implications. Faster backup replay carries positive security implications, e.g., a weakened verifier's dilemma and improved denial-of-service resilience, which we discuss in Section D.

Out of scope. Hints reveal storage-access patterns, public on Ethereum but unsuited to permissioned settings with confidential transactions. Identifying and penalizing peers that repeatedly deliver invalid hints, incentivizing correct hints, and compute-bound replay are deployment concerns we treat in Section D.

3 Ethereum: A case study

Before designing Ira-L, we conduct a systematic characterization of Ethereum state access patterns to understand the structure and locality of block execution workloads.

3.1 Methodology

To characterize Ethereum state access patterns, we re-executed two weeks of historical mainnet blocks and recorded every state operation performed by the EVM. Re-execution against archival state ensures deterministic, reproducible traces that capture the exact sequence of operations from original block validation.

Instrumentation. We instrumented *reth* [17], a production Rust Ethereum client, using its EVM Inspector interface. The Inspector intercepts every EVM instruction during execution, allowing us to capture state operations at the exact moment they occur. For each state-accessing opcode, we perform *stack introspection* to extract operand values (target addresses, storage slots) directly from the EVM stack before the operation executes.

We captured 14 distinct operation types: storage operations (`SLOAD`, `SSTORE`), account metadata queries (`BALANCE`, `SELFBALANCE`, `EXTCODESIZE`, `EXTCODEHASH`), bytecode access (`CALL`, `CALLCODE`, `STATICCALL`, `DELEGATECALL`, `EXTCODECOPY`), and state modifications (`CREATE`, `CREATE2`, `SELFDESTRUCT`). Table 1 shows the trace database schema.

Table 1: Trace database schema for EVM state operations.

Field	Type	Description
`block_number`	uint64	Block containing operation
`tx_index`	uint16	Transaction position in block
`op_index`	uint32	Operation position in transaction
`op_type`	uint8	Operation type (14 types)
`target_address`	bytes20	Contract or account accessed
`storage_slot`	bytes32	Storage slot (nullable)
`value_size`	uint32	Bytecode size (nullable)

Block Processing. For each block, we retrieve the parent block's state from reth's archival database and execute all transactions sequentially. To ensure correct intra-block state visibility, we maintain a *write-through cache* that accumulates state modifications across transactions within a block: each transaction observes the effects of all preceding transactions, matching original execution semantics. Blocks are processed in parallel across CPU cores, with results merged and sorted by $(block, tx, op)$ order.

Dataset. We collected traces from blocks 24,019,447 through 24,120,246 (100,800 consecutive blocks spanning December 15–29, 2025 on Ethereum mainnet). The trace contains 1,192,888,639 state operations across approximately 15 million transactions. We store traces in Apache Parquet [32] columnar format with Zstandard [29] compression, yielding a ∼12 GB dataset.

Completeness. Our instrumentation captures 100% of state operations executed during block validation, with no sampling or filtering. Transactions that fail during re-execution (e.g.,

due to state inconsistencies) are excluded; these represent less than 0.01% of transactions and do not affect our findings.

Implementation Note. Reth uses flat storage (the `PlainStorageState` table in MDBX, a high-performance key-value store) for execution-time state access, rather than direct Merkle Patricia Trie traversal as specified by the Ethereum Yellow Paper [71]. Each `SLOAD` or `SSTORE` corresponds to a single key-value lookup. Our traces capture these logical operations, which directly correspond to database I/O during block execution.

3.2 Findings

#1. I/O dominates execution. State I/O overwhelmingly dominates block execution time rather than EVM computation. Figure 3 shows the breakdown of execution time across 100,800 consecutive blocks: I/O accounts for 67.9% of total execution time, while pure EVM computation consumes only 32.1%. The median per-block I/O share is 66.8%, with the 5-95th percentile range spanning 55.6%–80.2%, indicating consistent I/O dominance across blocks regardless of transaction volume. We attribute to I/O the time spent inside backend state-access calls (account, storage, bytecode, and block-hash reads); all remaining time, including EVM opcode execution and accesses served from the in-memory cache, counts as compute.

This 2.1:1 of I/O-to-compute ratio has a critical implication: optimizing state access patterns yields far greater performance gains than optimizing EVM execution. Reduction in I/O latency through caching, prefetching, or sequential access patterns can significantly accelerate block validation.

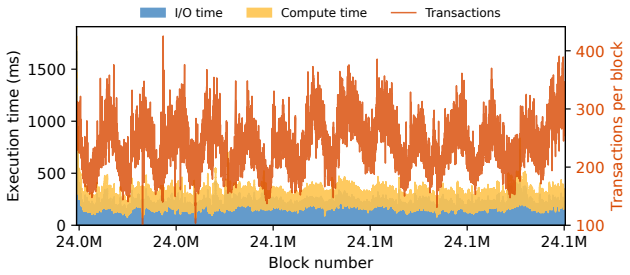


Figure 3: Execution time breakdown showing I/O dominance. The stacked area shows I/O time (blue) and compute time (yellow) per block; the line shows transaction count. I/O consistently dominates regardless of block size.

#2. Storage dominates I/O. Table 2 shows the distribution of state operations in our trace. Storage operations (`SLOAD` and `SSTORE`) account for 75.6% of all state accesses. Call operations, while numerous, contribute minimal I/O overhead: contract bytecode is loaded once per contract and cached in memory, so repeated calls to the same contract incur no disk I/O. Account metadata queries comprise less than 1% of operations.

Table 2: State operation distribution across 100,800 blocks.

Category	Operations	Share
Storage (`SLOAD`/`SSTORE`)	901,775,607	75.6%
Calls (bytecode load)	281,933,575	23.6%
Account metadata	6,751,623	0.6%
Creation/destruction	2,427,834	0.2%
Total	1,192,888,639	100%

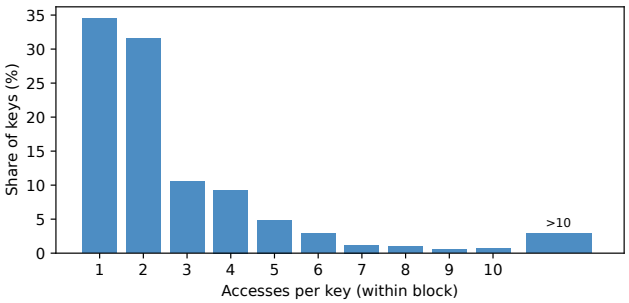


Figure 4: Intra-block access frequency distribution. Most keys are accessed 1–2 times per block, but a long tail of frequently-accessed keys drives high locality.

The read-write ratio for storage operations is 3.4 : 1 (696,683,837 `SLOAD`s vs. 205,091,770 `SSTORE`s), indicating that reads dominate the workload. Since reads benefit directly from cache hits while writes must persist regardless, this ratio suggests significant caching potential.

#3. High intra-block locality. We define the *reuse factor* as the ratio of total storage operations to unique storage keys accessed. Globally, the reuse factor is 25.74× (901,775,607 operations across 35,035,731 unique keys). However, this conflates intra-block and cross-block reuse.

To isolate intra-block locality, we examine the distribution of per-key access counts within blocks. Figure 4 shows this distribution: 34.5% of keys are accessed only once within their block, while 65.5% are accessed multiple times. The distribution exhibits a long tail, with some keys accessed up to 23,406 times within a single block.

This finding has important implications: we observe that 77.7% of key accesses within a block are to keys already accessed earlier in that block (i.e., of 901,876,518 storage accesses, only 200,839,090 are first accesses). A cache warmed with the correct keys before execution begins can serve most accesses as hits, thus underscoring the importance of an efficient caching system to improve block execution.

#4. Extreme Access Concentration. Access patterns exhibit

extreme concentration at both the contract and key levels. Figure 5 shows the cumulative storage access share as a function of contract rank: the top 3 contracts (USDT, USDC, and Compound) account for 42.2% of all storage operations, and the top 10 contracts account for 63.6%.

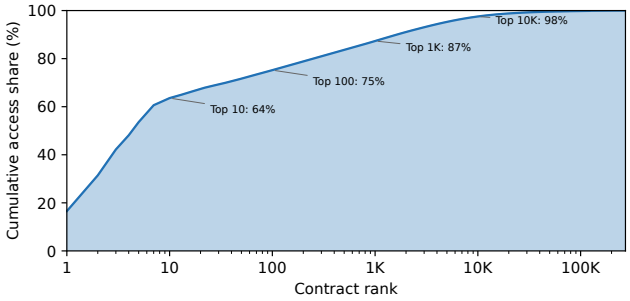


Figure 5: Cumulative storage access concentration by contract rank (out of 273,223 total contracts). A small number of contracts dominate state access: the top 10 contracts account for 63.6% and the top 10,000 account for 97.5% of all storage operations.

At the key level, concentration is even more pronounced. The top 1.23% of keys (430,873 keys with 100+ accesses) account for 84.0% of all storage operations. Conversely, 64% of keys are accessed only once or twice, contributing just 3.6% of total accesses. This concentration might suggest that caching the most frequently accessed keys globally would suffice. However, as Finding #5 shows, 72.4% of keys are ephemeral (appearing in only one block), and consecutive-block overlap averages just 12.4%. The "hot" keys change unpredictably from block to block, making static caching ineffective.

#5. Ephemeral Key Dominance. Despite high intra-block reuse, most keys are ephemeral and appear in only one block. Figure 6 shows the distribution of key lifespans: 72.4% of keys appear in only a single block, and 21.4% appear in 2–5 blocks. Only 1.0% of keys persist across more than 50 blocks.

Consecutive block overlap, i.e., the fraction of keys in block $n+1$ that also appeared in block n, averages only 12.4%, has a median of 11.7%, and a max of 100.0%. This low overlap implies that standard cache replacement policies like LRU cannot effectively predict which keys will be needed in upcoming blocks: LRU retains recently-used keys, but with only 12.4% overlap, most cached keys will not be accessed in the next block, while most keys needed will not be in cache.

#6. Bounded Working Set. Figure 7 summarizes per-block statistics. The median block performs 8,629 total operations, of which 6,587 are storage operations accessing 1,806 unique keys. Even at the 95th percentile, blocks access only 3,920 unique keys.

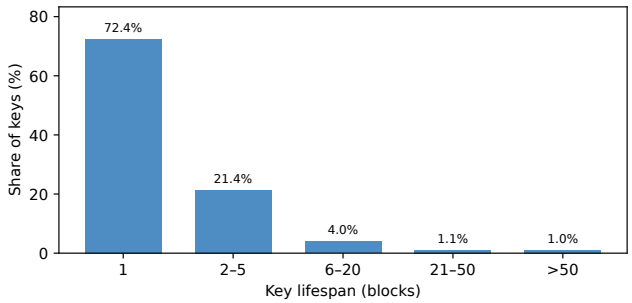


Figure 6: Key lifespan distribution: most keys are ephemeral and appear in only one block (72.4%), while less than 1.0% persist across more than 50 blocks.

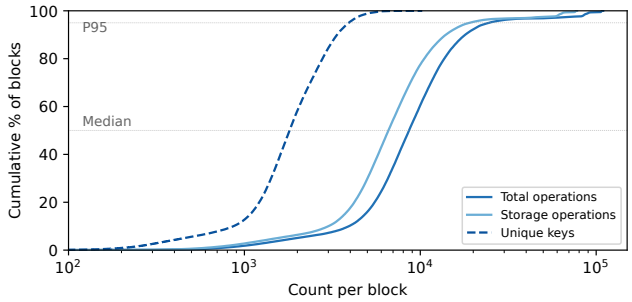


Figure 7: Cumulative distribution of per-block statistics. The working set remains bounded: 50% of blocks access fewer than 1,806 unique keys, and 95% access fewer than 3,920.

This bounded working set implies that per-block hints are tractable: hinting up to 2,000 keys requires at most 101 KB uncompressed (52 bytes per key: 20-byte address + 32-byte slot), which can be compressed significantly. The majority of blocks (58%) access fewer than 2,000 unique keys, and 84% access fewer than 3,000. Only 1.2% of blocks exceed 5,000 unique keys. This tight distribution suggests that a fixed-size hint buffer of 2,000–3,000 keys would cover the vast majority of blocks, making per-block prefetching hints both practical and efficient.

Takeaway. Our analysis reveals a fundamental tension. On one hand, I/O dominates execution (67.9%), storage dominates I/O (75.6%), and per-block working sets are bounded (median 1,806 keys), making caching critical and per-block hints tractable. On the other hand, 72.4% of keys are ephemeral and consecutive-block overlap averages just 12.4%, so LRU and other heuristic policies cannot predict which keys the next block will need. Ira-L (Section 4) closes this gap by transmitting the primary's access set to backups.

4 Ira-L: Latency-Optimal Algorithm

Ira-L instantiates the Ira framework (Section 2.3) for Ethereum block replay. After executing block b, the primary constructs a *hint* H_b containing three components: storage keys with source annotations, account addresses, and bytecode addresses (defined in Section 4.1). The primary transmits H_b alongside the block data. The backup uses H_b to prefetch all required state *before* execution begins, eliminating cold misses entirely.

We define the cold-miss barrier formally. Let A_b denote the set of unique storage keys accessed during execution of block b. Without hints, the backup discovers each key in A_b only at the moment it is first referenced, incurring I/O latency on every first access. Because I/O dominates execution time (2.1:1 I/O-to-compute ratio), eliminating these misses through prefetching reduces backup execution time toward the pure-compute lower bound. Section 5 confirms that backup execution closely approaches this bound, achieving a median 24.9× per-block speedup over the baseline.

4.1 Hint Generation

Key Collection. During execution of block b, an EVM inspector intercepts all state-accessing opcodes and records each accessed key into three sets:

- *Storage keys*: $(address, slot)$ pairs from `SLOAD` and `SSTORE`.
- *Account addresses*: from `BALANCE`, `SELFBALANCE`, `EXTCODESIZE`, and `EXTCODEHASH`.
- *Bytecode addresses*: from `CALL`, `STATICCALL`, `DELEGATECALL`, `EXTCODECOPY`, `CREATE`, and `CREATE2`.

Additionally, transaction senders, recipients, and the block beneficiary (coinbase) are added to the account set, since the EVM accesses these accounts implicitly during fee processing and reward distribution. The inspector is lightweight: it reads stack values at each relevant opcode without modifying execution behavior.

Source Annotation. For each storage key (a,k), the primary determines its *source*, which is a single-byte annotation indicating where the backup should read the value:

- **PlainState** ($\sigma = 0$): The current value resides in the main state table. A single B-tree seek suffices.
- **Zero** ($\sigma = 1$): The slot was never written. The value is zero. No disk I/O is required.
- **Changeset** ($\sigma = 2$): The value must be read from historical change sets. The full two-step lookup is required.

Without source annotations, the backup must perform a two-step lookup for every key: first seeking the history index to locate the value, then reading it from the appropriate table. The source annotation eliminates the first step for Plain State keys, reducing two seeks to one and eliminating all I/O for Zero keys. Across our dataset, 46.6% of storage keys resolve to PlainState, 10.5% to Zero, and 43.0% to Changeset. In total, 57.0% of keys bypass the history index entirely, with a per-block median of 55.3%. The source hint removes the confirmation step for PlainState keys and all I/O for Zero keys; only Changeset keys require the full two-step lookup.

Algorithm 1 Ira-L Hint Generation (Primary)

Require: Block $B = \langle T_1, \ldots, T_n \rangle$, State database $\mathcal{D}$
Ensure: Hint set H_B for block B

$S \leftarrow \emptyset$; $A \leftarrow \emptyset$; $C \leftarrow \emptyset$ // Storage, accounts, code
// Phase 1: Execute and collect accessed keys
for $i \leftarrow 1$ **to** n **do**
  Execute T_i with instrumented EVM
  $S \leftarrow S \cup \{(a,k) : T_i$ accesses storage slot k of account $a\}$
  $A \leftarrow A \cup \{a : T_i$ accesses balance or nonce of $a\}$
  $C \leftarrow C \cup \{a : T_i$ accesses bytecode of $a\}$
// Phase 2: Annotate storage keys with source information
for $(a,k) \in S$ **do**
  $b^* \leftarrow$ largest $b < B.num$ where (a,k) was modified, or $\perp$
  if $b^* = \perp$ **then**
    $src \leftarrow$ ZERO // Slot never written; value is zero
  else if b^* is in pruned history **then**
    $src \leftarrow$ PLAIN // Read from plain state table
  else
    $src \leftarrow$ HIST // Read from historical changeset
  $S[a,k] \leftarrow (a,k,src)$
// Phase 3: Serialize and compress
$H_B \leftarrow$ COMPRESS(S,A,C)
return H_B

Serialization. The hint is serialized in a compact binary format: each storage key requires 53 bytes (20-byte address + 32-byte slot + 1-byte source annotation), and each address key requires 20 bytes. Addresses and slots are fixed-width big-endian byte arrays, so the format is byte-order-explicit and portable across client, OS, and ISA. The serialized hint is compressed with Zstandard [29] and stored in a key-value database indexed by block number. Section 5 reports that the median compressed hint is 46.5 KB, achieving a 2.17× compression ratio which is negligible relative to the ∼2 MB block payload.

Algorithm 1 summarizes the hint generation procedure. The primary executes each transaction with the inspector attached (Phase 1), annotates every storage key with its optimal source (Phase 2), and serializes the result (Phase 3).

4.2 Hint-Based Replay

Sorted Batch Prefetching. Our key algorithmic contribution on the backup side is *sorted batch prefetching*. Upon receiving

hints for a batch of blocks, the backup proceeds as follows:

1. Collects all storage keys from the batch into a single sorted collection, ordered lexicographically by $(address, slot)$.
2. Routes keys by source annotation: PlainState keys are read via a single forward cursor walk through the B-tree, converting random I/O into a sequential scan; Zero keys are resolved immediately to zero with no I/O; Changeset keys are deferred to on-demand lookup during execution.
3. Similarly loads account metadata and bytecodes via sorted cursor scans.

Sorting is critical because the underlying database (MDBX, an LMDB-derived B-tree store) stores data in sorted key order. Sorted access maximizes page-cache locality: the cursor advances monotonically through the B-tree, touching each page at most once. Without sorting, each random seek may traverse $O(\log N)$ pages. By batching keys from multiple blocks (default batch size: 32 blocks), the amortized cost per key approaches a single sequential read.

Pipelined Execution. The backup operates in two phases:

- *Phase 1 (warm-up):* Parallel prefetch of approximately 3,000 blocks into an 8 GB in-memory buffer, using a thread pool for initial catch-up.
- *Phase 2 (steady-state):* A dedicated prefetcher thread reads hints and prefetches state in sorted batches, feeding populated caches to the executor through a bounded channel. The executor processes blocks sequentially, consuming prefetched state from the channel. The batch size is tuned so that the next batch completes before the executor finishes the current one, hiding prefetch latency behind execution.

This pipelining ensures that the backup's wall-clock time is bounded by $\max(\text{prefetch}, \text{execution})$ rather than their sum.

Memory Discipline. Each block's cache is discarded after execution and there is no cross-block cache. Since hints provide complete coverage for each block, retaining state across blocks is unnecessary. This bounds memory usage to a single block's working set and avoids the complexity of LRU eviction.

Algorithm 2 summarizes the backup replay procedure. The backup decompresses the hint, sorts all keys for sequential access (Phase 1), prefetches state via forward cursor traversal (Phase 2), and executes all transactions from the fully populated cache (Phase 3).

4.3 Correctness and Safety

Hint Completeness. Since we generate hints by instrumenting the actual EVM execution on the primary, they capture exactly the set of keys that will be accessed during replay. The deterministic nature of EVM execution guarantees that the backup, given the same block and state, will access the same keys.

Algorithm 2 Ira-L Replay (Backup)

Require: Block $B = \langle T_1, \ldots, T_n \rangle$, Hints H_B, State database $\mathcal{D}$

Ensure: Execution matches primary

```
(S, A, C) ← DECOMPRESS(H_B)
cache ← ∅
// Phase 1: Sort keys for sequential database access
S' ← SORTLEXICOGRAPHIC(S)                    // Sort by (a, k)
A' ← SORT(A);   C' ← SORT(C)
// Phase 2: Prefetch state via sequential cursor traversal
for a ∈ A' do                                // Sequential scan
    cache[a] ← D.READACCOUNT(a)
for (a, k, src) ∈ S' do                      // Sequential scan
    if src = ZERO then
        cache[a, k] ← 0                      // No I/O required
    else if src = PLAIN then
        cache[a, k] ← D.READPLAIN(a, k)
    else
        cache[a, k] ← D.READHIST(a, k, B.num)
for a ∈ C' do
    cache[a.code] ← D.READBYTECODE(a)
// Phase 3: Execute from fully-populated cache
for i ← 1 to n do
    Execute T_i using cache                  // All reads hit cache
```

Advisory Hints. The advisory property established in Section 2.3 manifests in Ira-L through Zstandard's built-in integrity check on hint decompression and graceful fallback to reth's standard execution path on any coverage gap. Section D discusses adversarial-hint detection and mitigation. Section 2.3 states the full adversary model.

State Verification. After executing each block, the primary computes a deterministic hash over all state changes which includes accounts, storage slots, and bytecodes, sorted by key for reproducibility. The backup can optionally verify that its execution produced an identical hash, providing end-to-end correctness checking without requiring full Merkle state root computation.

5 Evaluation

5.1 Configuration

Hardware. Apple M1 Pro (10-core: 8 performance + 2 efficiency), 32 GB unified memory. Benchmarks and trace databases on external 8 TB NVMe SSD (Sabrent, USB enclosure).

Software. Rust 1.88 (edition 2024), Python 3.14 with matplotlib 3.10, numpy 2.4. Apache Parquet (Arrow 54) with Zstandard compression for trace storage. reth (commit 62abfdae) for trace collection (requires synced Ethereum mainnet

Table 3: Primary overhead breakdown. Wall time compares end-to-end execution; hint cost decomposes the overhead into construction and serialization as a fraction of aggregate execution time (15,717 s).

Component	Time (s)	% of Exec Time
Baseline wall time	23,250	—
Primary wall time	29,889	—
Hint construction	940	6.0%
Hint serialization	778	5.0%
Hint total	1,719	10.9%
Wall-time overhead	+28.6%	

node).

Methodology. We evaluate Ira-L using the same two-week dataset of 100,800 consecutive Ethereum mainnet blocks (blocks 24,019,447–24,120,246). All benchmarks start with a cold OS page cache (purged before each run) to measure worst-case I/O performance. We compare four configuration families:

- *Baseline*: unmodified reth executing blocks sequentially.
- *Primary*: reth with hint generation enabled.
- *Backup (sequential)*: backup replaying with hints using a single prefetch thread.
- *Backup (parallel-k)*: backup replaying with k concurrent prefetch threads ($k \in \{1,2,4,6,8,10,12,16,32,64\}$). We report detailed per-block results for $k = 16$ and $k = 64$; the full sweep is used in the thread-scaling analysis (Section 5.4).

5.2 Hint Overhead on Primary

The primary generates hints during normal block execution. We measure two components of overhead: *hint construction* (collecting the access set) and *hint serialization* (compressing and writing hints to disk). We compare against the unmodified baseline to quantify the total cost.

Wall-Time Overhead. Table 3 summarizes the overhead. The baseline executes all 100,800 blocks in 23,250 s, while the primary completes in 29,889 s—a 28.6% increase. Of the hint-related work, construction (collecting the access set and querying the history index) accounts for 940 s (6.0% of aggregate execution time) and serialization (compressing and writing to disk) accounts for 778 s (5.0%), totaling 1,719 s (10.9%). The remaining wall-time increase reflects secondary effects of instrumentation (e.g., memory pressure, cache interference).

Per-Block Hint Cost. Figure 8 shows the per-block distribution of hint cost as a fraction of execution time. The median per-block hint fraction is 12.1%, with a 95th-percentile of 22.1%. Although construction accounts for more aggregate time than serialization (Table 3), the serialization CDF has a heavier right tail: serialization time is roughly constant (∼12.5 ms), so it dominates the per-block fraction for short-executing blocks, while construction scales with block size and its fraction remains bounded. Each CDF is sorted independently, so the total-cost curve is the distribution of per-block sums, not the pointwise sum of the two component curves.

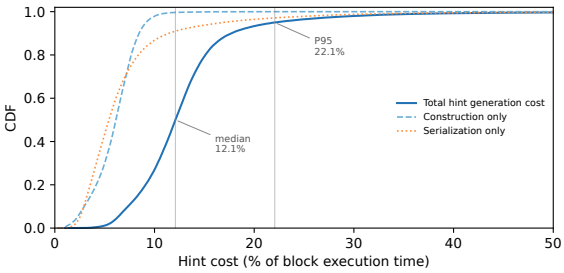


Figure 8: CDF of per-block hint cost as a percentage of block execution time. Each curve is an independent distribution: the total-cost curve is the CDF of per-block $(t_{\text{construct}} + t_{\text{serialize}})/t_{\text{exec}}$, not the sum of the two component CDFs. Median total hint fraction is 12.1%; P95 is 22.1%.

In absolute time, construction spans a wide range (median 14.4 ms, P95 33.8 ms) and scales linearly with the number of unique storage keys ($r = 0.79$, ∼8.1 µs/key). Serialization clusters tightly around ∼12.5 ms regardless of block size ($r = 0.03$), dominated by the fixed cost of flushing the database write transaction. Construction therefore dominates aggregate cost (Table 3) even though serialization dominates the per-block *fraction* for short-executing blocks. The per-block hint fraction varies primarily because *execution time* varies—driven by I/O and page cache state—not because hint cost itself is unpredictable.

These per-block fractions (median 12.1%, P95 22.1%) reflect a conservative, fully sequential measurement where hint construction and serialization run inline with block execution. In practice, both components are independent of execution correctness and can be overlapped with the next block's processing. Hint construction (6.0% of aggregate execution time; Table 3) builds a map from keys already collected during EVM execution and queries the storage history index to annotate each key with its optimal read source. Hint serialization (5.0%) compresses and persists the hint to disk, requiring no executor state at all. Production reth already employs a multi-threaded architecture with a dedicated executor thread pool and a separate storage writer thread; in this model, hint construction can execute on the existing thread pool and serialization can run on the storage thread, overlapping both with the next block's execution. Under such pipelining, the effective overhead approaches zero for all but the final block in a batch. Moreover, the cost is paid once by the primary and recovered on every backup: even with sequential prefetching, the backup replays 5.2× faster in wall time (Section 5.3),

yielding a net system-wide speedup despite the primary's 28.6% wall-time increase.

Hint Sizes. Figure 9 shows the distribution of per-block hint sizes. The median compressed hint is 46.5 KB (mean 51.5 KB), with a 95th-percentile of 104.2 KB and a maximum of 239.0 KB. Zstandard compression achieves a 2.17× reduction from the raw representation (median 102.2 KB raw). Figure 10 confirms that these sizes remain stable across the entire 100,800-block range, with no visible trend or drift.

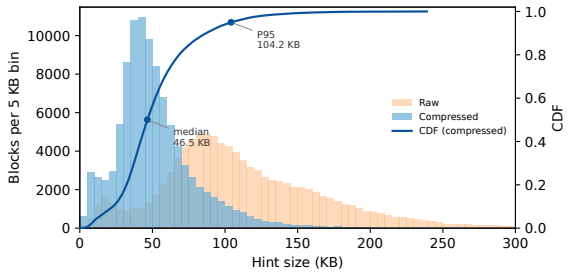


Figure 9: Distribution of per-block hint sizes. Histogram bins show block counts for raw (orange) and compressed (blue) sizes; the overlaid CDF (dark blue) marks the median (46.5 KB) and P95 (104.2 KB) of compressed hints.

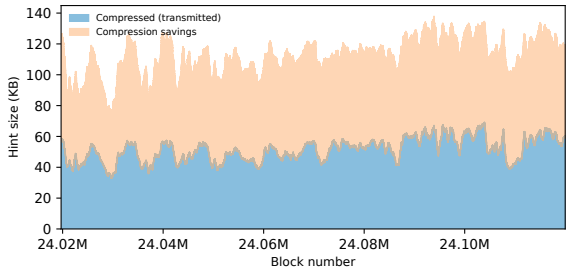


Figure 10: Hint sizes across blocks (500-block rolling average). The bottom layer shows the compressed size transmitted to backups; the top layer shows the bytes eliminated by Zstandard compression.

A hint contains one fixed-size entry per unique storage key accessed in the block, encoding the key hash and its optimal read source. The raw hint size is therefore determined entirely by the key count: a block touching 1,806 unique keys (the median; Section 3) produces a hint of 102.2 KB before compression. Zstandard is effective because, while the key hashes themselves are high-entropy, the accompanying metadata fields (source flags, offsets) are highly regular.

These hint sizes are negligible relative to the block data they accompany (∼2 MB per block): even the 95th-percentile compressed hint adds only ∼5% to the block payload. On a 1 Gbps link, transmitting the P95 hint takes under 1 ms which is orders of magnitude less than the replay time saved. Hint size is also bounded by the EVM gas limit, which caps the number of storage operations per block. In practice, hints grow only when blocks are large, and large blocks are precisely those where replay savings are greatest (Section 5.3).

5.3 Backup Speedup

We now evaluate how effectively hints accelerate backup replay. For each block, the backup reads the hint, prefetches the indicated keys into its MDBX cache, and then executes the block. We report the per-block speedup as speedup $= t_{\text{baseline}}/(t_{\text{wait}} + t_{\text{exec}})$, where t_{baseline} is the baseline execution time, t_{wait} is the time the executor stalls waiting for prefetch, and t_{exec} is the backup's execution time.

5.3.1 Per-Block Speedup Distribution

Figure 11 shows per-block speedup across the entire block range for the sequential backup configuration; the full version [24] tabulates the key percentiles for all three configurations.

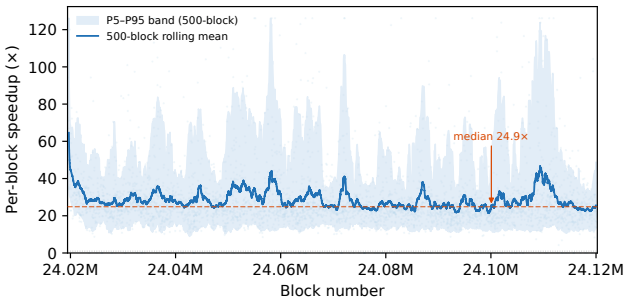


Figure 11: Per-block speedup (sequential backup vs. baseline) over the 100,800-block trace. Each point is one block; the solid line is a 500-block rolling mean and the shaded band spans the rolling P5–P95 within each window. The dashed line marks the overall median (24.9×).

The backup achieves a median *per-block* speedup of 24.9× (mean 28.9×) across the sequential configuration, with the parallel configurations yielding nearly identical distributions. In terms of aggregate wall time, the sequential backup completes all 100,800 blocks in 4,366 s (5.2× faster than the baseline's 22,604 s), parallel-16 in 956 s (23.6×), and parallel-64 in 871 s (25.9×). The gap between per-block and wall-time speedup for the sequential configuration arises because the 3.02% of blocks with regressions (Section 5.3.2) contribute disproportionately to aggregate wall time.

The speedup magnitude reflects the I/O-dominance of baseline execution (Section 3): because storage I/O accounts for the majority of block execution time, prefetching the entire working set before execution begins eliminates nearly all of it, leaving only EVM computation.

Figure 11 reveals three regimes. First, an *initial transient* in the first ∼500 blocks where speedup exceeds 50×: the baseline starts with a fully cold OS page cache, while the backup begins against an already-populated MDBX file and completes with zero wait time. As the baseline's page cache fills, its execution time drops and the speedup converges to steady state.

Second, the *steady-state* regime (blocks 1,000+) exhibits a stable rolling mean of 27-30× with no meaningful drift and dividing the trace into 10 equal segments yields segment means that vary by less than ±10% of the overall mean. This stability confirms that the speedup is determined by the per-block I/O fraction rather than by transient cache effects.

Third, *periodic bursts* of 60-130× appear throughout the trace. These high-speedup blocks have 2× higher baseline execution time (more page cache misses) and 38% more unique storage keys than the median, yet complete with zero backup wait time. The bursts are not sustained, the median cluster length is 1 block. This indicates that they arise from individual blocks whose working sets happen to miss heavily in the baseline's page cache.

This 5.2× single-thread speedup exceeds the ∼3× that Amdahl's law predicts from the 67.9% I/O share. That bound assumes the remaining 32.1% is fixed compute. In practice, much of it is time spent servicing in-memory cache hits during execution, which Ira changes fundamentally: it prefetches the working set into a flat, sorted vector, so these accesses also run faster through improved memory locality. Ira accelerates both the I/O and the compute terms, so a bound derived from the I/O share alone does not apply.

5.3.2 Tail Behavior

While the median speedup is substantial, we examine the fraction of blocks where the backup is *slower* than the baseline.

With sequential prefetching, 3.02% of blocks experience a regression. The root cause is MDBX's use of memory-mapped I/O: each cold page access blocks the faulting thread until the kernel completes the read, so a single prefetch thread issues at most one storage I/O at a time. When a block touches thousands of uncached B-tree pages, this serialization causes the executor to stall waiting for prefetch to finish, negating the I/O savings. Among these regressed blocks, the median speedup is 0.18× (5.5× slower than baseline), with a mean of 0.20× (4.9× slower). At 64 prefetch threads, only 0.09% of blocks remain slower than the baseline.

5.3.3 Wait Time Analysis

Figure 12 decomposes aggregate wall time by whether the executor stalled waiting for prefetch to complete.

With sequential prefetching, 96.5% of blocks complete with zero wait time, yet the remaining 3.5% blocks, which are the same large-working-set blocks identified in Section 5.3.2,

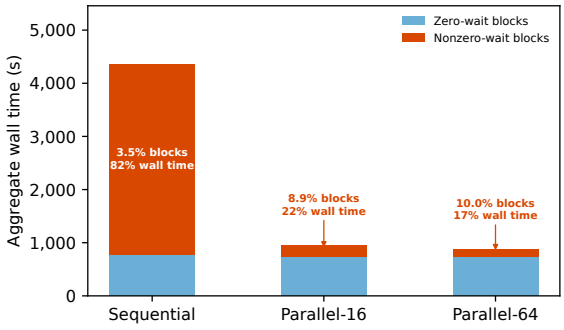


Figure 12: Aggregate wall time decomposed by executor wait. For each configuration, the bottom segment is wall time from blocks with zero wait; the top segment is from blocks where the executor stalled for prefetch.

account for over 80% of aggregate wall time. Parallel prefetching eliminates these stalls: with multiple threads, page faults are in flight simultaneously and the prefetch phase completes before the executor is ready to begin. This reduction in wait time is the primary driver of the wall-time improvement from sequential to parallel configurations, as execution time itself is unchanged.

5.4 Thread Scaling

Figure 13 shows the wall-time speedup relative to the baseline as a function of prefetch thread count across ten configurations (1–64 threads). The curve exhibits three regimes. First, a *steep scaling* region from 1 to 8 threads where wall time drops from 4,366 s to ∼1,088 s (4× reduction), reflecting near-linear conversion of serial page faults into parallel I/O. Second, a *diminishing-returns* region from 8 to 16 threads where wall time falls further to 956 s (4.6× over sequential) as the SSD's command queue approaches saturation. Third, a *plateau* beyond 16 threads: doubling from 16 to 32 yields only a 7% reduction, and quadrupling to 64 threads reaches 871 s (1.1× over 16 threads), indicating that the I/O bandwidth of our NVMe SSD is fully saturated.

The steep initial gains reflect the conversion of serial page faults into parallel I/O: when a single thread prefetches sequentially, each MDBX page fault blocks until the SSD services the request. With multiple concurrent threads, page faults are in flight simultaneously, saturating the SSD's queue depth and achieving higher throughput. The knee at 8-16 threads corresponds to the device's maximum queue depth being reached; additional threads merely compete for the same bandwidth. The optimal thread count is hardware-dependent since systems with higher-bandwidth storage (e.g., PCIe 5.0 NVMe, RAID arrays) may shift the knee to higher thread counts.

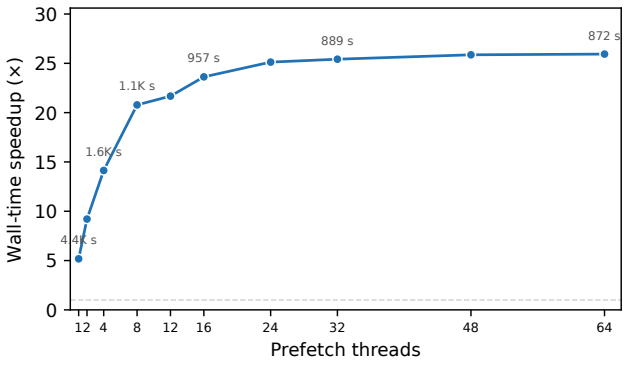


Figure 13: Wall-time speedup relative to the baseline as a function of prefetch thread count (1–64 threads). Labels show absolute wall time. The curve saturates around 8–16 threads, consistent with I/O bandwidth limits of the NVMe SSD.

5.5 Adversarial Hints

We measure the time for the latency of execution caused by a malformed hint. An adversary that sends a hint of the same size as the honest one but with non-existing keys, so accesses during execution miss the prefetched cache. In our system, this triggers the fallback to standard replay (Section D). We measure this cost with respect to the baseline cost of a system that nodes run today.

Methodology. We replay 1,000 randomly sampled blocks against the reth state with a cold OS page cache, purged before each pass as in Section 5.1. Each block is replayed two ways: *baseline*, standard reth with no hints and on-demand reads; and *malicious*, a parallel sorted-batch prefetch of the spurious hint followed by the fallback replay. The spurious hints match the per-block size of the captured hints (median 2,696 keys across the 100,800 captured blocks), and prefetching uses 16 threads. We report the per-block slowdown as the ratio of malicious to baseline replay time.

Results. Under the worst-case malicious hint, the backup replays at a median of 63.5 ms per block against the baseline's 41.1 ms, a median slowdown of 1.58× (mean 2.12×). The tail stays bounded at 5.5× (P95) and 8.3× (P99). A malicious hint slows 81.5% of blocks relative to baseline, confirming that the adversary imposes a real but small cost rather than an unbounded one.

Analysis. The standard-replay fallback reads exactly the state the backup would have read without Ira, so a malicious hint adds at most one wasted prefetch to an otherwise standard replay. This is akin to a peer relaying an incorrectly signed block or a block with incorrect metadata: the backup detects the invalid data during replay and discards the hint, falling back to the path it would have taken without Ira. Standard peer-management mechanisms that drop or penalize peers on receiving invalid data limit the impact further, since a primary or relay that repeatedly delivers bad hints is disconnected. The slowdown therefore stays bounded and never affects correctness. We discuss detection and peer-level mitigation further in Section D.

6 Related Work

Optimal Cache Replacement. Belady's MIN algorithm [21], proved optimal by Mattson et al. [52], provides optimal offline cache replacement by evicting the item whose next access is furthest in the future. Some works use adaptive heuristics, e.g., the *adaptive replacement cache* [53] (ARC) and LIRS [43] which use heuristics to dynamically balance between recency and frequency, and LeCaR [70] uses regret minimization to learn this balance online. Another approach is to use hardware prefetchers which predict future accesses to hide memory latency. Prefetch-aware replacement policies like PACMan [73] and SHiP [72] adjust eviction priorities for prefetched blocks. A different approach is to approximate the MIN algorithm from history. The Hawkeye cache [41] uses past access patterns to approximate MIN decisions. Jain and Lin [42] later observe that MIN minimizes *total* misses but not *demand* misses in the presence of prefetching, introducing Demand-MIN and Flex-MIN to navigate this tradeoff. Subsequent works like Mockingjay [63] and Glider [64] use machine learning to improve prediction accuracy but require specialized hardware structures and still cannot exceed the accuracy ceiling imposed by predicting from past behavior alone. Recent deep learning approaches [33,65,76] use neural networks to learn access patterns, achieving higher accuracy than traditional heuristics but remaining fundamentally limited to inference from history. Our work differs from these works in that we transmit exact future access information from primary to backup, enabling true optimal eviction without prediction.

Hint-based systems. Mowry et al. [56] presented the pioneering work with the idea of using compiler based automation to optimize prefetching. The TIP system [59] by Patterson et al. was one of the early works proposing the *hint*-based paradigm where the applications advise the OS with *advisory hints* to improve prefetching and caching optimizations. HintStor [34] presents a framework to propagate I/O hints across the storage stack in heterogeneous storage systems. Mandal et al. [51] demonstrate hints at the block layer of storage for deduplication, where applications mark unique writes (`NODEDUP`) and prefetch requests (`PREFETCH`) to avoid redundant hash computations. Modern OS provide system calls such as `madvise` [5] and `fadvise` [6] where applications can advise the OS on the access pattern such as `random`, `sequential`, `will-need`, and `don't need`. Database systems such as Oracle's `INDEX` and `USE_NL` hints allow developers to override the query planner run before executing a query. These approaches share

the limitation that hints are *manually* specified and require domain knowledge, or *statically derived* which are limited to predictable patterns like loops. In contrast, our framework Ira generates hints automatically from the primary's execution trace, providing complete and exact future access information without programmer effort.

Replay in Replicated Databases. Modern databases build on ARIES-style write-ahead logging [55], which established the three-phase recovery protocol (analysis, redo, undo) now used across the industry [54]. In primary-backup replication, backups continuously replay the primary's log to maintain synchronized state. Deterministic databases such as Calvin [67] and Aria [49] ensure replicas converge to identical states by replaying transaction logs in a deterministic order without coordination. A key challenge is the *parallelism gap*: primaries execute transactions concurrently, but backups traditionally replay logs serially. KuaFu [28] addresses this by tracking dependencies to enable parallel replay, demonstrating that serial replay sustains less than one-third of primary throughput thus underscoring the need for systems such as Ira. Cloud-native databases have further evolved replay architectures: Amazon Aurora [69] pushes redo log application to a distributed storage tier, treating the log as the database itself, while Socrates [20] (Azure SQL Hyperscale) uses a four-tier architecture with dedicated Page Servers for log application. PolarDB-SCC [75] tackles the consistency challenges arising from asynchronous log propagation to read-only replicas. While these works focus on replay correctness, parallelism, or architectural placement of log application, none optimize *cache eviction decisions* during replay. Ira is complementary: it accelerates replay within any of these architectures by exploiting the primary's knowledge of future accesses to achieve Belady-optimal caching.

Blockchain State Synchronization. Blockchain nodes face significant replay challenges during both initial synchronization and ongoing block validation. Modern Ethereum clients use staged sync architectures [1,17] that pipeline header, body, and execution stages; despite these optimizations, the execution stage remains the primary bottleneck. To enable parallel validation, some systems require transactions to declare data dependencies upfront: Ethereum's EIP-2930 [61] introduced optional access lists, though adoption remains low [38]; Solana's Sealevel runtime [74] mandates read/write set declarations. For systems without mandatory declarations, Block-STM [35] provides optimistic parallel execution, and Forerunner [27] exploits speculative pre-execution. While these works optimize sync architecture or transaction scheduling, none address *cache eviction* during replay. Unlike optional, user-declared access lists, Ira's hints are execution-derived, complete by construction, and carry a per-key source annotation (Section 4.1) that eliminates history-index seeks, enabling optimal caching during the execution stage. Section E provides extended discussion.

Speculative and parallel execution. Parallel transaction execution has been extensively studied. Kung and Robinson's optimistic concurrency control (OCC) [46] established the read-validate-write paradigm; Silo [68] demonstrated OCC can achieve nearly 700,000 transactions per second on multi-core hardware. For high-contention workloads, Doppel [57] introduced phase reconciliation, and deterministic approaches like BOHM [30] pre-compute dependencies to enable parallel execution. Ira is orthogonal to these approaches: even when transactions execute in parallel, each thread must manage its cache. By providing exact future access information, Ira enables optimal caching *within* each parallel execution context. Section E provides extended discussion.

7 Conclusion

We presented Ira, a general framework for accelerating backup replay in primary-backup replication by transmitting compact hints that encode the primary's knowledge of future access patterns. The framework applies to any storage-bound, deterministic primary-backup system including replicated databases, distributed key-value stores, and blockchain platforms. We demonstrated its practicality through Ira-L, a concrete instantiation for Ethereum block synchronization implemented in the production client reth, showing significant replay speedups at low hint overheads.

References

[1] Erigon/eth/stagedsync/README.md at release/2.60 · erigontech/erigon. URL: https://github.com/erigontech/erigon/blob/release/2.60/eth/stagedsync/README.md.

[2] Eth/63 fast synchronization algorithm by karalabe · Pull Request #1889 · ethereum/go-ethereum. URL: https://github.com/ethereum/go-ethereum/pull/1889.

[3] Geth v1.10.0. URL: https://blog.ethereum.org/2021/03/03/geth-v1-10-0.

[4] Introduction | Monad Developer Documentation. URL: https://docs.monad.xyz/.

[5] Madvise(2) - Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/madvise.2.html.

[6] Posix_fadvise(2) - Linux manual page. URL: https://www.man7.org/linux/man-pages/man2/posix_fadvise.2.html.

[7] Specs/pubsub/gossipsub/gossipsub-v1.1.md at master · libp2p/specs. URL: https:

//github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.1.md.

[8] The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure, August 2022. URL: https://aptosnetwork.com/whitepaper/aptos-whitepaper_en.pdf.

[9] 26.2. Log-Shipping Standby Servers, November 2025. URL: https://www.postgresql.org/docs/18/warm-standby.html.

[10] Certification-Based Replication | Galera Cluster | MariaDB Documentation, September 2025. URL: https://mariadb.com/docs/galera-cluster/galera-architecture/certification-based-replication.

[11] Cosmos Stack Developer Documentation, 2025. URL: https://docs.cosmos.network.

[12] MySQL :: MySQL 8.0 Reference Manual :: 20 Group Replication, 2025. URL: https://dev.mysql.com/doc/refman/8.0/en/group-replication.html.

[13] The Sui Smart Contracts Platform. 2025.

[14] Erigontech/erigon. Erigon, January 2026. URL: https://github.com/erigontech/erigon.

[15] Ethereum/devp2p. ethereum, February 2026. URL: https://github.com/ethereum/devp2p.

[16] Ethereum/go-ethereum. ethereum, January 2026. URL: https://github.com/ethereum/go-ethereum.

[17] Paradigmxyz/reth. Paradigm, January 2026. URL: https://github.com/paradigmxyz/reth.

[18] Sei Parallelization Engine: Multi-core Blockchain Execution | Sei Docs, January 2026. URL: https://docs.sei.io/learn/parallelization-engine.

[19] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, April 2018. URL: http://arxiv.org/abs/1801.10228, arXiv:1801.10228.

[20] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1743–1756, New York, NY, USA, June 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3299869.3314047.

[21] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. URL: http://ieeexplore.ieee.org/document/5388441/.

[22] Adithya Bhat. Adithyabhatkajake/ira-rs, May 2026. URL: https://github.com/adithyabhatkajake/ira-rs.

[23] Adithya Bhat, Mohsen Minaei, and Harshal Bhadreshkumar Shah. Ira: Efficient Transaction Replay for Ethereum (Artifact Data), May 2026. URL: https://zenodo.org/doi/10.5281/zenodo.20262361.

[24] Adithya Bhat, Harshal Bhadreshkumar Shah, and Mohsen Minaei. Ira: Efficient Transaction Replay for Distributed Systems, January 2026. URL: https://arxiv.org/abs/2601.21286v1.

[25] 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. Sui Lutris: A Blockchain Combining Broadcast and Consensus, October 2024. URL: http://arxiv.org/abs/2310.18042, arXiv:2310.18042.

[26] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. pages 1–14, November 2019. URL: http://arxiv.org/abs/1807.04938, arXiv:1807.04938.

[27] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based Speculative Transaction Execution for Ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 570–587, New York, NY, USA, October 2021. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3477132.3483564.

[28] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. KuaFu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1186–1195, Brisbane, QLD, April 2013. IEEE. URL: http://ieeexplore.ieee.org/document/6544908/.

[29] Facebook. Zstandard - Real-time data compression algorithm. URL: https://facebook.github.io/zstd/.

[30] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control, December 2015. URL: http://arxiv.org/abs/1412.2324, arXiv:1412.2324.

[31] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5):613–624, January 2017. URL: https://dl.acm.org/doi/10.14778/3055540.3055553.

[32] {Apache Software Foundation. Apache Parquet. URL: https://parquet.apache.org/docs/overview/.

[33] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3311–3322, November 2020. URL: https://ieeexplore.ieee.org/abstract/document/9211554.

[34] Xiongzi Ge, Zhichao Cao, David H. C. Du, Pradeep Ganesan, and Dennis Hahn. HintStor: A Framework to Study I/O Hints in Heterogeneous Storage. *ACM Trans. Storage*, 18(2):18:1–18:24, March 2022. URL: https://dl.acm.org/doi/10.1145/3489143.

[35] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. *arXiv preprint arXiv:2203.06871*, 2022. arXiv:2203.06871.

[36] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, June 1996. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/233269.233330.

[37] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. Nurgle: Exacerbating Resource Consumption in Blockchain State Storage via MPT Manipulation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2180–2197, May 2024. URL: http://arxiv.org/abs/2406.10687, arXiv:2406.10687.

[38] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, Roger Wattenhofer, and Patrick Züst. Dissecting the EIP-2930 Optional Access Lists, December 2023. URL: http://arxiv.org/abs/2312.06574, arXiv:2312.06574.

[39] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, August 2020. URL: https://dl.acm.org/doi/10.14778/3415478.3415535.

[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, June 2010. USENIX Association.

[41] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News*, 44(3):78–89, October 2016. URL: https://dl.acm.org/doi/10.1145/3007787.3001146.

[42] Akanksha Jain and Calvin Lin. Rethinking Belady's Algorithm to Accommodate Prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 110–123, June 2018. URL: https://ieeexplore.ieee.org/document/8416822/.

[43] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002. URL: https://dl.acm.org/doi/10.1145/511399.511340.

[44] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, USA, June 2011. IEEE Computer Society. URL: https://doi.org/10.1109/DSN.2011.5958223.

[45] Erin Kenneally and David Dittrich. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research, August 2012. URL: https://papers.ssrn.com/abstract=2445102, arXiv:2445102.

[46] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981. URL: https://doi.org/10.1145/319566.319567.

[47] John Kuszmaul. Verkle Trees, 2018. URL: https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf.

[48] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/.

[49] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment*, 13(12):2047–2060, August 2020. URL: https://dl.acm.org/doi/10.14778/3407790.3407808.

[50] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying Incentives in the Consensus Computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, October 2015. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2810103.2813659.

[51] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using Hints to Improve Inline Block-layer Deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, 2016. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/mandal.

[52] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. URL: http://ieeexplore.ieee.org/document/5388318/.

[53] Nimrod Megiddo and Dharmendra S. Modha. {ARC}: A {Self-Tuning}, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003. URL: https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache.

[54] C Mohan. Repeating History Beyond ARIES. *VLDB*, 99:7–10, September 1999.

[55] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. URL: https://dl.acm.org/doi/10.1145/128765.128770.

[56] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Not.*, 27(9):62–73, September 1992. URL: https://dl.acm.org/doi/10.1145/143371.143488.

[57] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/narula.

[58] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. *Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.*, June 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[59] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 79–95, New York, NY, USA, December 1995. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/224056.224064.

[60] Daniel Perez and Benjamin Livshits. Broken Metre: Attacking Resource Metering in EVM, March 2020. URL: http://arxiv.org/abs/1909.07220, arXiv:1909.07220.

[61] Ethereum Improvement Proposals. EIP-2930: Optional access lists. URL: https://eips.ethereum.org/EIPS/eip-2930.

[62] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. URL: https://dl.acm.org/doi/10.1145/98163.98167.

[63] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective Mimicry of Belady's MIN Policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572, April 2022. URL: https://ieeexplore.ieee.org/document/9773195.

[64] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-52, pages 413–425, New York, NY, USA, October 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3352460.3358319.

[65] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 861–873, New York, NY, USA, April 2021. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3445814.3446752.

[66] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1493–1509, New York, NY, USA, May 2020. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3318464.3386134.

[67] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, May 2012. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2213836.2213838.

[68] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, November 2013. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2517349.2522713.

[69] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, May 2017. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3035918.3056101.

[70] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 3, USA, July 2018. USENIX Association.

[71] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, pages 1–32, 2014.

[72] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441, Porto Alegre Brazil, December 2011. ACM. URL: https://dl.acm.org/doi/10.1145/2155620.2155671.

[73] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, and Joel Emer. PACMan: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 442–453, Porto Alegre Brazil, December 2011. ACM. URL: https://dl.acm.org/doi/10.1145/2155620.2155672.

[74] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain. 2025.

[75] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proceedings of the VLDB Endowment*, 16(12):3754–3767, August 2023. URL: https://dl.acm.org/doi/10.14778/3611540.3611562.

[76] Yiyuan Yang, Rongshang Li, Qiquan Shi, Xijun Li, Gang Hu, Xing Li, and Mingxuan Yuan. SGDP: A Stream-Graph Neural Network Based Data Prefetcher, October 2023. URL: http://arxiv.org/abs/2304.03864, arXiv:2304.03864.

[77] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, July 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3293611.3331591.

[78] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar

Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2653–2666, New York, NY, USA, June 2021. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3448016.3457559.

A Ethical Considerations

Scope. Ira-L's main contribution is a performance optimization for honest Ethereum block replay, accompanied by a security analysis of its trade-offs (Sections 2.3 and 5.5). Hints are advisory and independently verified by the backup (Sections 2.3 and 4.3): a primary capable of proposing a malicious block before Ira-L has the same capability after, and a backup that ignores hints reproduces the unmodified baseline. We structure the analysis below using the Menlo principles [45].

Stakeholders. Four groups are affected: (i) Ethereum node operators (validators, RPC providers, archive nodes); (ii) block proposers that emit hints; (iii) Ethereum users whose transactions are validated; and (iv) the broader ecosystem (protocol developers, client implementers, researchers).

Respect for Persons. Experiments use only public mainnet data collected on our own archive node. No private data, no human subjects, no identifying information about validator-running entities. Transactors are pseudonymous, and the storage-access patterns our hints encode are revealed on-chain after confirmation, so the hint channel adds no new information about identifiable persons. Adoption is opt-in: no stakeholder must interact with Ira-L to remain a valid participant.

Beneficence. We weigh benefits against bounded harms per stakeholder.

- *Operators* gain a 23.6× backup speedup (Section 5.3), lowering the commodity-hardware bar for running a full or archive node. Adversarial hints cause bounded slowdown, mitigated by standard-replay fallback (Section 4.3); an adopting node is never worse off than a non-adopting one. We rejected authoritative (non-advisory) hints because a Byzantine proposer could then corrupt backup state.
- *Proposers* pay 10.9% in hint-generation overhead (Section 5.2) and gain no MEV, fee, or ordering capability. Malformed hints waste backup CPU but are bounded and discarded at the gossip layer, analogous to invalid block headers or bodies.
- *Users* benefit from a tighter attestation window and a weakened verifier's dilemma [50], with no direct harm and no new privacy surface.
- *Ecosystem*: the client-agnostic hint format (Section 4.1) and open reference implementation make the benefits portable. An asymmetric rollout could widen the short-run cost gap between hint-aware and unmodified nodes, mitigated by releasing the implementation alongside the paper.

Negative applications considered. We flag two:

- *Cheaper on-chain surveillance*: a lower archive-node cost also lowers the cost of large-scale address-activity indexing for actors hostile to transactor privacy. On balance, well-resourced surveillance actors already have this capability, and the decentralization gain for honest operators outweighs the marginal reduction.
- *Hint-gossip side channel*: timing the arrival of hints could in principle leak storage-access sets earlier than today's block gossip, whose propagation is already an observable channel studied for validator deanonymization and topology inference [**?**, **?**]. The signal is weak (hints come only from the primary, and block contents are revealed within seconds regardless).

Justice. Benefits are not restricted to any operator class. The design is client-agnostic, the implementation is openly licensed, and the hint format has no hardware, OS, or ISA dependency (Section 4.1). Evaluation singles out no individual or organization; all statistics are aggregated over the 100,800 public mainnet blocks studied (Section 3).

Respect for Law and Public Interest. We use only public mainnet data and circumvent no protocol rule, client license, or applicable regulation. No vulnerability disclosure is triggered. The public-interest case is the decentralization and consensus-security gain. The artifact accompanies the paper [23].

Decision. Under Beneficence, the benefits (lower replication cost, stronger consensus-layer security through a weakened verifier's dilemma, broadly distributed access) exceed the bounded and mitigatable harms (adversarial-hint slowdown, short-run rollout gap, marginal surveillance cost). A Respect-for-Persons analysis reaches the same conclusion: interaction is only via public on-chain data, no consent violation, no new Byzantine capability granted to the primary. We determine that publication is appropriate.

B Open Science

We are committed to reproducibility. The following artifacts are available:

Available Artifacts.

- *Source code*: Our modifications to reth implementing hint generation (primary) and hint-based prefetching (backup), along with trace collection instrumentation. Available at [22].
- *Analysis scripts*: Python scripts for computing all statistics and generating all figures in this paper.
- *Benchmark results*: CSV files containing raw timing data for all experiments (hint overhead, backup speedup, thread scaling).
- *Trace statistics*: Aggregated statistics from our workload analysis (operation counts, key distributions, per-block

metrics) in CSV format.

The data artifacts (pre-generated hints, Parquet traces, and state hashes) are archived at https://doi.org/10.5281/zenodo.20262361.

Artifacts Not Shared.

- *Ethereum archival node*: Reproducing trace collection requires a fully synced Ethereum archival node (∼3 TB). We cannot redistribute this data due to its size, but it is freely available to anyone running an archival node.
- *Raw execution traces*: The Parquet trace files (∼12 GB) are derived from publicly available blockchain data. We provide aggregated statistics and the collection scripts; researchers with archival access can regenerate the traces.

Reproducibility. The repository includes a `README.md` with instructions for: (1) building the modified reth client, (2) running benchmarks using pre-generated hints, and (3) regenerating all figures from the provided CSV data. Researchers without archival node access can reproduce all evaluation results using our pre-computed hint database and benchmark CSVs.

C Generalization to a Second Temporal Window

To check that Ira-L's speedup generalizes, we re-ran the full pipeline of Section 5 (identical binaries, hardware, and cold-cache protocol) on blocks 20,850,000–20,950,000 (100,001 blocks, September 28 to October 12, 2024, 14 months earlier). The median per-block speedup is 9.0× (parallel-16), aggregate wall-clock speedup is 10.1×, primary overhead is 7.3%, and only 0.07% of blocks regress. These are uniformly lower than the December 2025 headline (24.4×, 23.6×, 10.9%), and the gap is a workload effect: re-running the trace pipeline of Section 3 on the October window yields a median of 3,894 storage operations per block (1.69× below December's 6,587) and 1,211 unique storage keys (1.49× below December's 1,806), yet median baseline execution time is 4.5× lower (45.3 versus 203.0 ms). Protocol-priced work scaled 1.69× while wall-clock I/O cost scaled 4.5×, reflecting state-trie growth that lengthens cold-page lookups; Ira-L's amplification tracks this I/O fraction, so the heavier later baseline yields the larger speedup. The same mechanism delivers an order-of-magnitude acceleration in either window.

D Discussion

Application scope. Our framework has applications beyond Ethereum. The key requirements for Ira to be applicable in a distributed database system are: (i) primary-backup architecture, i.e., there exists a node that executes first and the others follow, (ii) replay is storage-bound, where I/O is a bottleneck, and (iii) the execution is deterministic. Systems such as the following potentially benefit from our solution:

1. *Replicated databases*, e.g., MySQL Group Replication [12], MariaDB Galera [10], PostgreSQL streaming replication [9], CockroachDB [66], TiDB [39], and deterministic databases like Calvin [67], use primary-backup architecture to deterministically replay database operations. All of them have nodes with complete execution traces that can assist replicas during replication.

2. *Distributed key-value stores*, e.g., FoundationDB [78], use consensus-based replication where followers replay the primary's logs. For stores with large state that exceeds memory, hints can naturally encode access patterns to improve performance.

3. *Blockchain systems* such as Solana [74], Sui [13], Aptos [8, 35], Cosmos [11, 26], and the Hyperledger family [19] have validators that propose blocks while other nodes replay using a VM layer, and are strong candidates for Ira's hint-based acceleration. Layer-2 rollups (sequencer as primary) and sharded chains (per-shard primary-backup replay) are natural fits. Ira needs no single leader either: in leaderless DAG-based protocols such as Narwhal [**?**] and Bullshark [**?**], any validator that has executed a batch can gossip its hints as an advisory byproduct, and access sets from different proposers are unionable.

Deployment considerations. *Layer-1 optional.* Hint correctness need not be enforced by the underlying synchronization protocol, whether in database replication or in blockchain block proposal.

Incentive Mechanisms. We can incentivize our system with rewards. Today, Ethereum's EIP-2930 access lists [61] incentivize transactions that declare an access list (a list of accounts accessed by the transaction) with discounted gas costs, though adoption remains low [38]. If the access list is wrong, the user pays more gas. A similar mechanism can ensure that block producers generate correct hints.

Security considerations. Our hint generation does not rely on the primary being non-faulty. On omission of hints, nodes can simply revert to a backup caching mechanism such as LRU. However, systems operating under stronger trust models, e.g., those using a Trusted Execution Environment (TEE) or crash-fault models, can take advantage of the non-faulty primary and provide additional hints that can further speed up verification as well as replay. Note that hints reveal access patterns, which may be a privacy consideration in permissioned blockchains where transaction contents are confidential.

Security implications. Faster backup replay carries two security consequences. First, it narrows the verifier's dilemma [50]: cheaper re-execution raises a rational validator's incentive to verify each block, making honest validation the dominant

strategy. Second, it improves resilience to denial-of-service: slow state access is a known attack surface, whether through mispriced execution [60] or deliberate state-storage bloat [37], and Ira-L's lower replay latency shrinks the window such attacks exploit. Both consequences follow from the optimization and remain orthogonal to its safety guarantees.

Corrupted hints. A peer delivering a corrupted or malicious hint is analogous to a peer delivering a corrupt block: in both cases, the recipient detects the fault and discards the invalid data. For hints, detection is straightforward: decompression failure (Zstandard integrity check), a malformed entry count, or a cache miss during execution all cause the backup to discard the hint and fall back to standard replay. Because hints are advisory and never affect state correctness, the impact of a corrupted hint is bounded to a single block's replay latency. Two further mechanisms bound this cost: the backup can heuristically upper-bound the number of storage accesses from the block's gas and cap the accepted hint size accordingly, rejecting oversized hints before prefetching; and prefetching hints in sorted batches confines I/O and cache pollution to that per-block working set, so even a maximally-spurious same-size hint cannot evict state beyond a single block's footprint. Our measurements confirm this bounds the slowdown to a small factor over standard replay (Section 5.5).

We treat the problem of *identifying and penalizing* peers that deliver corrupt hints as orthogonal to our work. Ethereum's networking stack already provides mechanisms for this: the consensus layer's gossipsub peer scoring [7] tracks peer behavior and disconnects peers whose scores fall below a threshold, while the execution layer's devp2p protocol supports explicit disconnect codes for protocol violations [15]. These same mechanisms can be extended to penalize peers that consistently deliver invalid hints, requiring no changes to Ira itself.

Limitations and future work. Our implementation targets a single-threaded execution model. Concurrent execution would require coordination for hint data structures, potentially through partitioning hints by key ranges, usage of synchronization primitives and barriers. Notably, Ira can also assist concurrent execution with hints that aid in improving concurrent replay. We can extend the hints to support parallel replay across multiple threads and explore partitioned hint structures to avoid synchronization overheads. Another limitation of our work is that it assumes the replay workload is storage-bound. Compute-bound replay, such as zk-proving or crypto-heavy opcodes, is out of scope and left to future work. Beyond cache eviction, hints could guide write batching, memory allocation, and as already stated, concurrency to further accelerate replay.

E Extended Related Work

Blockchain State Synchronization (Extended). Early Ethereum clients performed full sync by re-executing every transaction from genesis, but this approach became impractical as chain history grew. Geth [16] introduced fast sync [2] (2015) and later snap sync [3] (2021), which download recent state directly and reconstruct the Merkle trie locally, achieving 10× faster initial sync. Erigon [14] (formerly Turbo-Geth) pioneered *staged sync* [1], a pipelined architecture that processes headers, bodies, senders recovery, and execution as discrete stages; Reth [17] adopts this architecture in Rust for further performance gains. Despite these optimizations, the execution stage cannot be parallelized within staged sync.

Regarding access lists, analysis shows only 1.46% of transactions include access lists despite 42.6% benefiting from them [38]. Solana's Sealevel runtime [74] mandates read/write set declarations, enabling lock-based parallel execution across non-conflicting transactions. Block-STM [35] achieves 170k transactions per second on Aptos using software transactional memory techniques. Sui [25] takes an object-centric approach where owned objects bypass consensus entirely, enabling sub-second finality for non-shared state. Forerunner [27] exploits the time window between transaction dissemination and consensus to speculatively pre-execute transactions, achieving 10× speedup through constraint-based multi-trace specialization. Looking forward, Verkle trees [47] promise to enable stateless clients by reducing witness sizes from ∼150 KB to 1–2 KB, allowing validators to verify blocks without storing full state.

Speculative and Parallel Execution (Extended). For workloads with high contention on popular records, Doppel [57] introduced phase reconciliation, cycling between split phases (where commutative operations execute on per-core state) and joined phases (where results are merged), achieving up to 38× higher throughput than conventional protocols. Deterministic approaches like BOHM [30] pre-compute transaction dependencies to enable parallel execution without runtime conflict detection, while PWV [31] increases concurrency by making writes visible before transaction commit through piece-wise execution. In the blockchain space, commercial efforts such as Monad [4] and Sei [18] further pursue optimistic parallel EVM execution. These works share a common goal: maximizing transaction throughput by exploiting parallelism. Ira is orthogonal to and composable with all of these approaches.