Ira: Efficient Transaction Replay for Distributed Systems

Adithya Bhat Harshal Bhadreshkumar Shah Mohsen Minaei

Visa Research

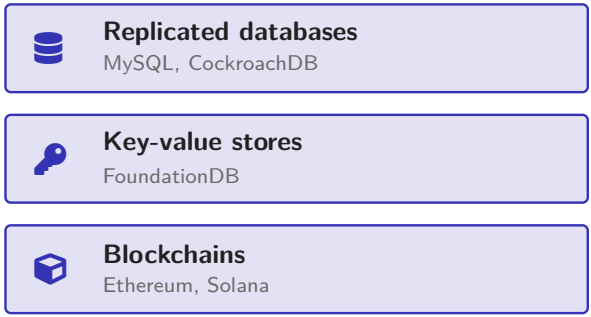
USENIX Security 2026

Talk Overview

1. **Ethereum Case Study**
2. Ira-L
3. Performance

Primary-Backup Replication

1. One node executes first, the rest replay
2. Databases, key-value stores and blockchains all use it
3. Replay is the cost they all share



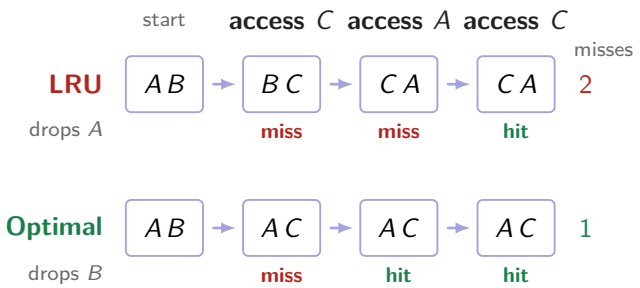
every one of them replays what another node already ran

Cache Eviction on the Backup

Every replay reads state from storage. Which key should a backup keep?

1. Cache holds two keys: A and B
2. The next accesses are C, A, C
3. LRU drops A, then needs A back

- **Whoever already ran the block knows the order**



Future Knowledge in Primary-Backup

Optimal eviction needs to know what comes next.

1. Optimal caching needs the upcoming accesses
2. A backup starting a batch does not have them
3. But the primary has already executed that batch

- **The primary holds exactly what the backup is missing**

In general



In primary-backup



has already run it

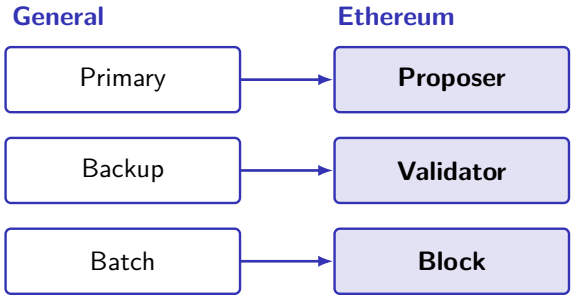
the backup is about to discover, one miss at a time, a sequence the primary already holds

Ethereum Is Primary-Backup Replication

One node always knows. Where does that happen at scale?

1. Ethereum is the largest decentralized system running today
2. Its *proposer* is the primary: it executes the block first
3. Its *validators* are the backups: each replays that same block

- **Same pattern, at global scale**



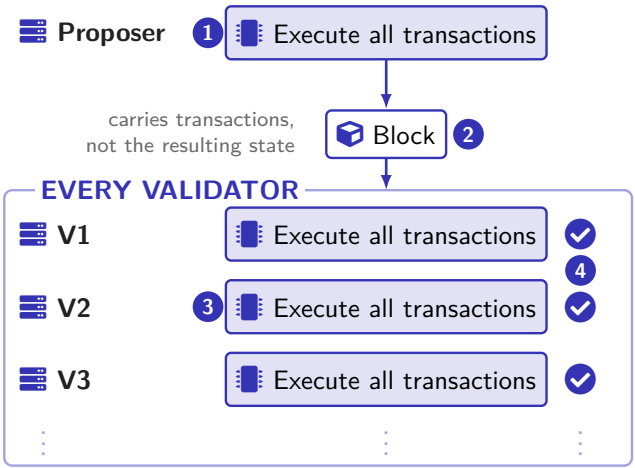
the words change, the structure does not

Ethereum Block Validation

Proposer executes, validators replay. Here is that in detail.

1. Proposer executes transactions
2. Block goes to every validator
3. Each validator replays
4. Each validator verifies state

- **Every validator redoes the proposer's work**

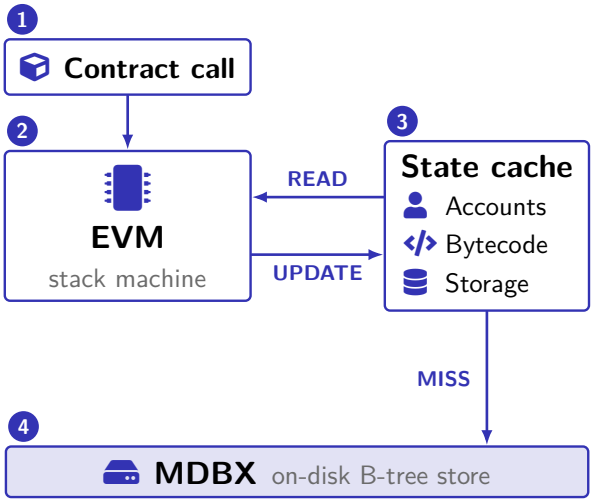


EVM Computation and State Access

Every validator redoes the proposer's work. What is that work?

1. Contract call enters the EVM
2. EVM executes bytecode
3. Reads hit the state cache
4. A miss falls through to MDBX

- **Every state read can fall to disk**



Measurement Dataset

That work is state access. Is it slow enough to matter? We measured.

- Instrumented reth's EVM

reth: the state-of-the-art Ethereum client

- Replayed two weeks of mainnet
- Recorded *all* state operations, in order
- All 14 op types · no sampling
- **Every state access of every block, in order**



100,800
CONSECUTIVE BLOCKS

24.05M
TRANSACTIONS

1.193B
STATE OPERATIONS

Blocks 24,019,447–24,120,246 · December 15–29, 2025

Execution-Time Breakdown

We captured every state access. First question: where does the time go?

OBSERVED RATIO

2.1:1

State I/O to compute

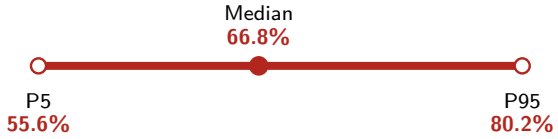
Even the quietest blocks spend most of their time on state: P5 is 55.6%

- **Replay is storage-bound**

AGGREGATE EXECUTION TIME



PER-BLOCK I/O SHARE



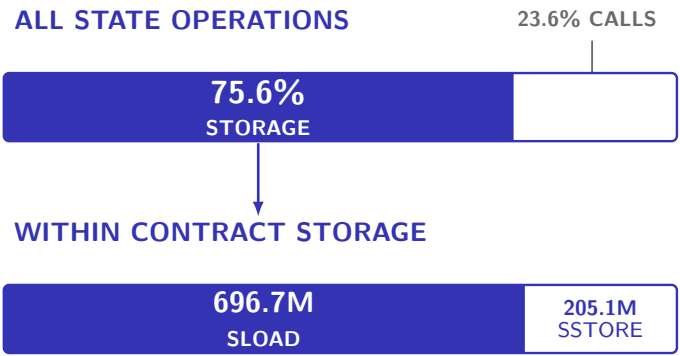
I/O = backend state-access calls
compute includes cache hits

State-Operation Distribution

Two thirds of replay is state I/O. Which operations is that?

1. 1.193B state operations
2. 75.6% contract storage
3. 3.4 reads per write

- **The state path is read-heavy**



3.4:1 reads to writes

Storage = SLOAD + SSTORE

Other 0.8% = account metadata, creation, destruction

Intra-Block Key Reuse

Storage reads dominate. Do those reads repeat?

1. Block × storage key
2. First touch to the key
3. Later touches reuse it

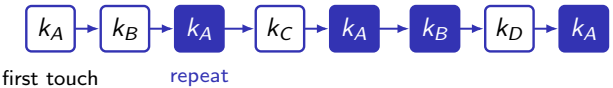
- **State is reusable within one block**

77.7%

Reusable after first touch

4.49× accesses per key

ONE BLOCK · ACCESS ORDER



ALL STORAGE ACCESSES



Repeat = key seen earlier in the same block

Touches Per Key

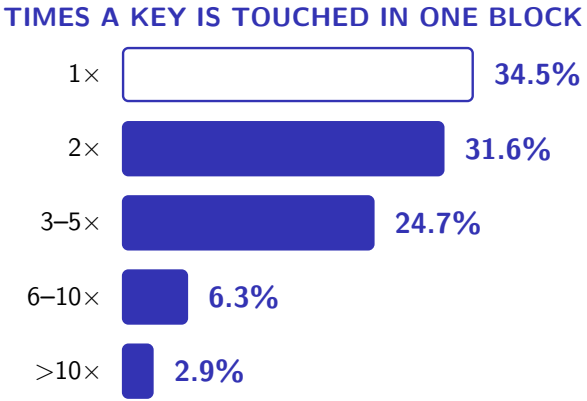
Accesses repeat. How many times is a key touched?

1. Count every touch to a key inside its block
2. Only 34.5% are touched once
3. 4.49× accesses per key

- **Most keys come back within the block**

65.5%

Keys touched more than once



65.5% of block-key pairs are touched at least twice

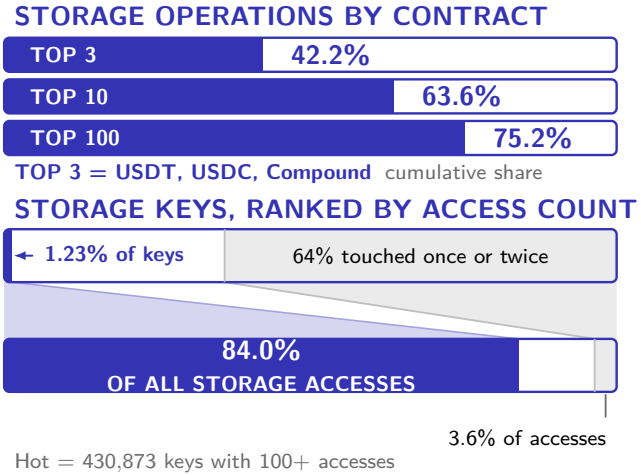
200,839,090 block-key pairs · up to 23,406 touches

Access Concentration

Most accesses re-touch a key from the same block. Is that reuse concentrated?

1. 273,223 contracts touched
2. USDT, USDC, Compound: 42.2% of accesses
3. Top 1.23% of keys: 84.0% of accesses

- **A tiny hot set covers most accesses**



History as a Predictor

A tiny hot set covers most accesses. So cache the hot keys?

1. Do the hot keys stay hot?
2. 72.4% of keys appear in one block only
3. Block n covers 12.4% of block $n{+}1$

Median 11.7% · P95 24.8%

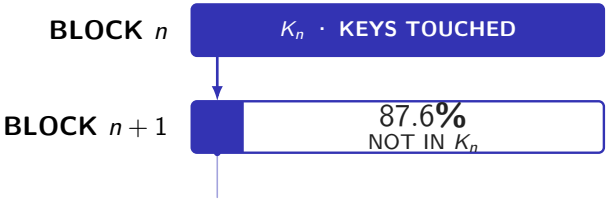
- **History cannot name the next block's keys**

2 HOW MANY BLOCKS TOUCH A KEY



6+ blocks: 6.1%, including 1.0% over 50
Blocks containing a key, not a consecutive span

3 ADJACENT BLOCK KEY SETS



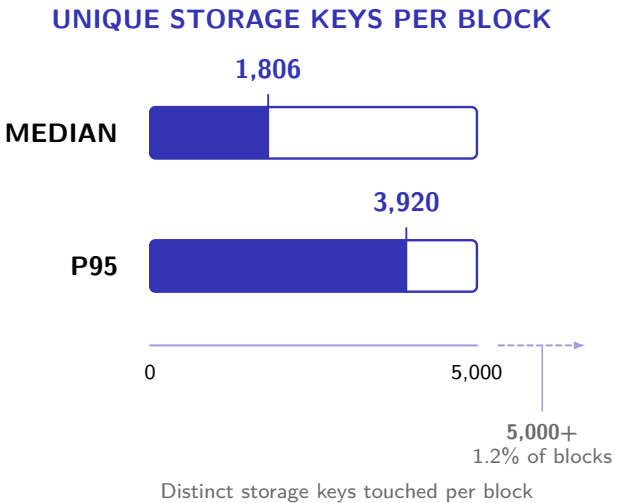
$|K_n \cap K_{n+1}|/|K_{n+1}|$ across 100,687 adjacent block pairs

Per-Block Working-Set Size

History cannot name the next block's keys, so they must be sent. How many?

1. Unique address–slot keys per block
2. Median 1,806 keys · P95 3,920 · max 10,258
3. 84% of blocks stay under 3,000 keys
4. ∼199 KiB of raw key identities at P95
5. 46.5 KB median once encoded and compressed

- **Small enough to ship with the block**



Case-Study Takeaways

Four measurements. Together they say one thing.

67.9%

2.1:1 vs. compute

I/O matters

77.7%

of accesses in a block

Reuse exists

12.4%

seen in the prior block

History cannot predict it

1,806

median keys per block

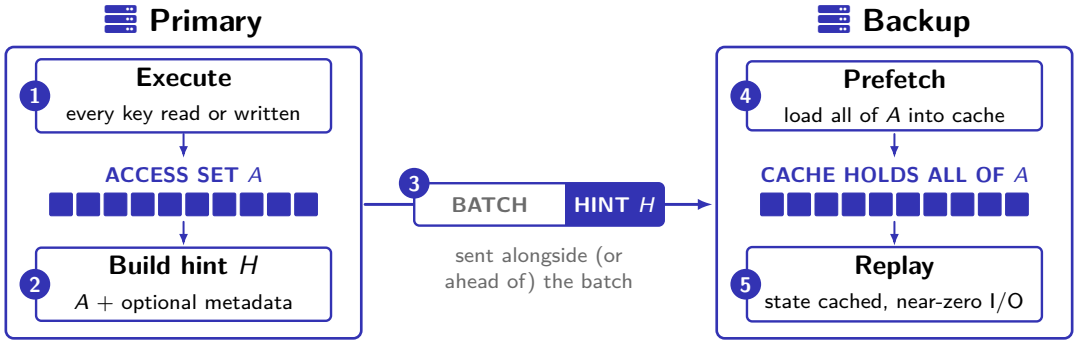
Exact set is tractable

- **Small, exact, and knowable before replay**

1. Ethereum Case Study
2. **Ira-L**
3. Performance

Ira: Hint-Based Replay

The exact set is knowable. Ira is the protocol that moves it.

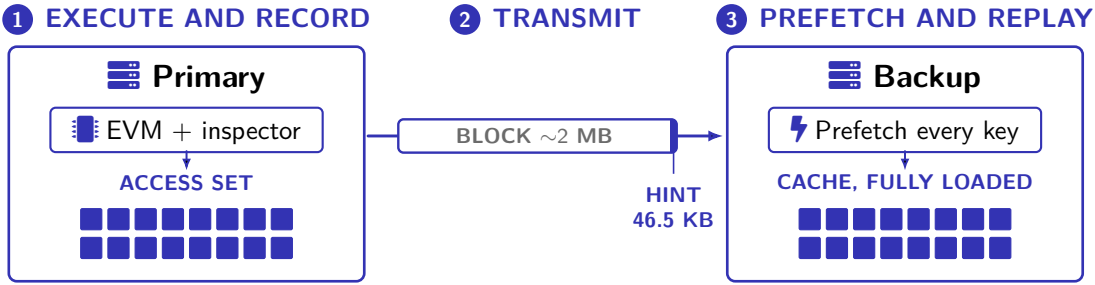


Hints are advisory: performance only, never correctness

missing, corrupt, or incomplete: the backup falls back to standard replay

- **Any primary-backup system can run these five phases**

Only the proposer knows it. So let it say so.



Record every key touched, tag its source, compress

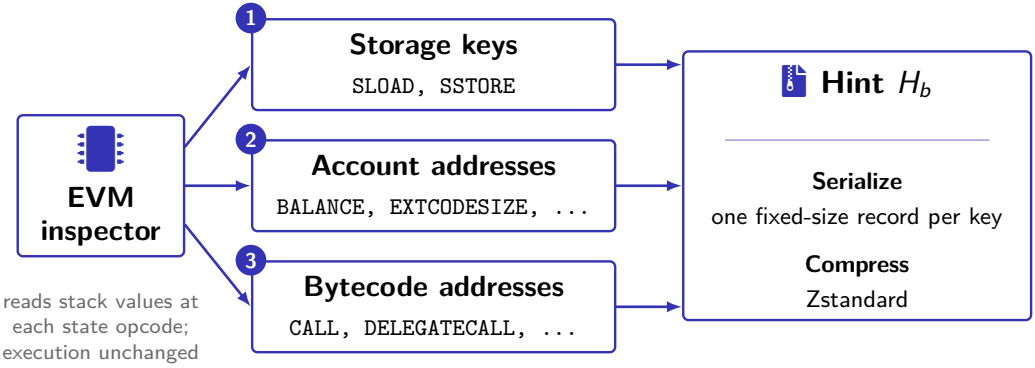
Ship the hint with the block the primary already sends

Pull the whole working set into cache, then execute

- **The backup loads its working set up front**

Hint Generation

The primary ships a hint alongside the block. What goes in it?



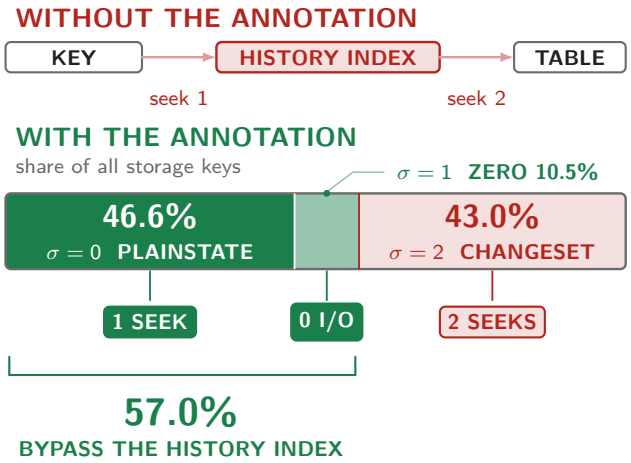
- **Hints fall out of execution already happening**

Source Annotations

The hint names every key. It also says where each one lives.

1. reth asks the history index which table to read
2. PlainState holds current values, Changeset holds superseded ones
3. σ answers that question in one byte

- **One byte removes an entire index lookup**



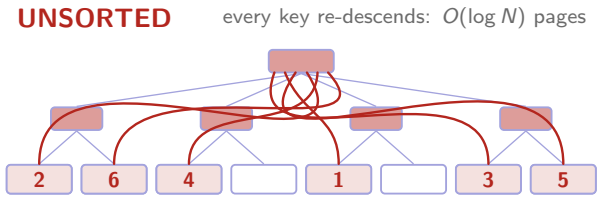
per-block median 55.3%
only Changeset keys still pay both seeks

Sorted Batch Prefetching

The backup knows which keys, and where. Now read them fast.

1. Sort every key in a 32-block batch
2. PlainState: one cursor walk
3. Zero: no I/O; Changeset: on demand
4. Accounts, bytecodes: same scan

- **Random seeks become one ordered sweep**



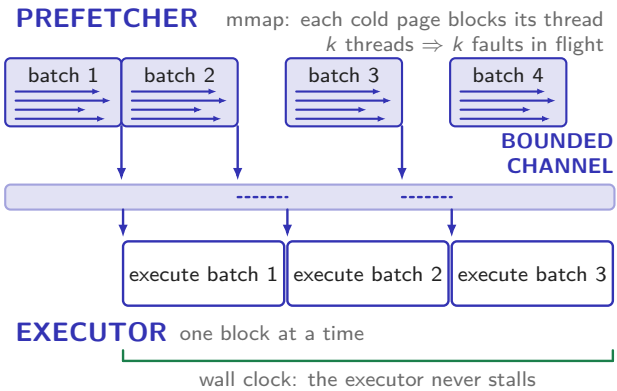
MDBX (LMDB-derived B-tree) stores pages in sorted key order

Pipelined Execution

Sorted reads are fast. Now hide them behind execution.

1. Prefetcher thread fills sorted batches
2. k threads keep k page faults in flight
3. Bounded channel carries populated caches
4. Next batch is ready before it is needed

- **Wall time is the larger of the two, not the sum**



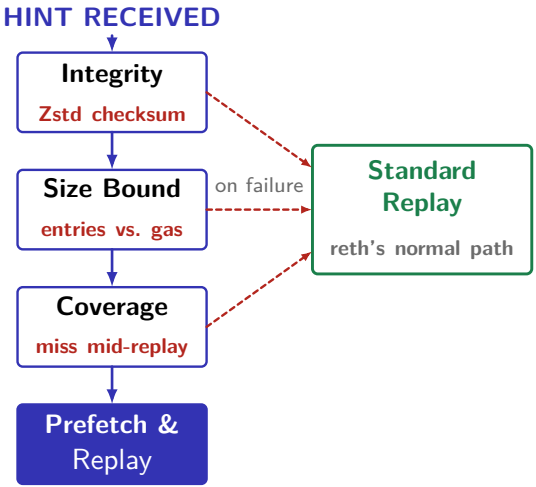
Memory: each block's cache is discarded after execution; no cross-block cache, no LRU

Hint Safety and Fallback

That is the mechanism. What if the hint is wrong?

1. Hints are advisory: they affect replay latency alone
2. Per-block batches cap a bad hint's cost at one block
3. A per-block hash lets backups verify each replay

- **A bad hint costs latency, never correctness**



Performance

1. Ethereum Case Study
2. Ira-L
3. **Performance**

Evaluation Setup

A bad hint costs latency, never correctness. Now: does it pay off?

- Ira-L in reth, the fastest-syncing Ethereum client (staged sync)
- Apple M1 Pro (8P+2E) · 32 GB · external 8 TB NVMe SSD
- Same 100,800 blocks as the case study (24,019,447–24,120,246)
- Cold OS page cache, purged before every run
- **Every number ahead is worst-case I/O**

CONFIGURATIONS

Baseline
unmodified reth, sequential

Primary
hint generation enabled

Backup (sequential)
1 prefetch thread

Backup (parallel-k)
k threads, 1–64

Detailed per-block results at $k = 16$ and $k = 64$

Backup Replay Performance

Cold cache, worst-case I/O. Start with the backup.

AGGREGATE WALL TIME

23.6×

Backup, 16 prefetch threads

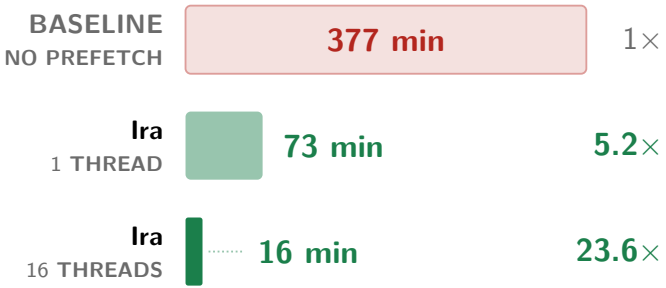
MEDIAN PER-BLOCK

24.4×

$t_{\text{base}}/(t_{\text{wait}} + t_{\text{exec}})$, so prefetch stalls count against Ira

- **Prefetching removes nearly all of replay's I/O**

WALL TIME OVER ALL 100,800 BLOCKS



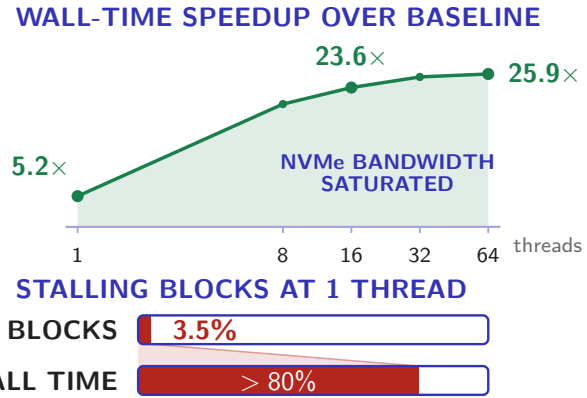
Amdahl on the 67.9% I/O share alone predicts ∼3×. But Ira prefetches the working set into a flat sorted vector, so the 32.1% compute term (mostly cache hits) speeds up too.

Prefetch Scaling and Tail Behavior

Fast on aggregate. Does every block win?

1. One prefetch thread: 5.2×
2. MDBX mmap: one thread issues one I/O at a time
3. Fault-heavy blocks stall the executor
4. Concurrent faults remove the stalls, up to SSD bandwidth

- **A few stalling blocks dominate wall time**



Blocks slower than baseline: 3.02% at 1 thread, 0.09% at 64
Knee is hardware-dependent: faster storage shifts it right

Primary Overhead and Hint Size

The backup wins big. What does the primary pay for it?

1. Hints built during normal execution
2. 10.9% of aggregate execution time
3. +28.6% wall time vs. baseline

- **Paid once, recovered on every backup**

5.2×

Faster backup replay, one prefetch thread

HINT WORK of 15,717 s execution



construct 6.0% (940 s) + serialize 5.0% (778 s)
= 1,719 s of 15,717 s aggregate execution

END TO END: +28.6% wall time
baseline 23,250 s → primary 29,889 s
gap over the 10.9% direct work: memory pressure, cache interference

HINT VS BLOCK PAYLOAD



median 46.5 KB · Zstd 2.17× from 102.2 KB raw
under 1 ms on a 1 Gbps link

Adversarial Hints

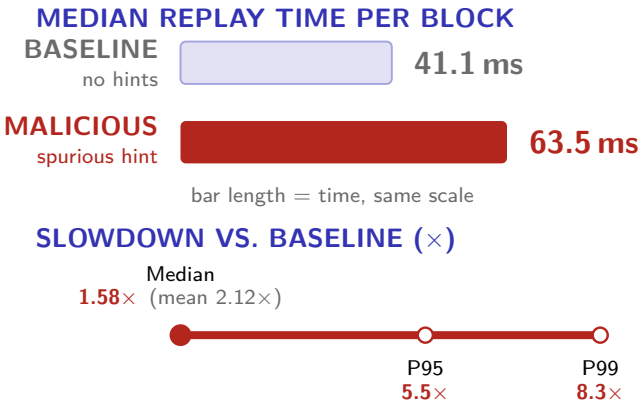
Cheap when the primary is honest. What if it lies?

1. Byzantine primary: same-size, spurious hint
2. 1,000 blocks, cold cache, baseline vs. malicious replay
3. Fallback reads exactly what the baseline would

- **A bounded cost, never a wrong answer**

Correctness unchanged

state root always verified



81.5% of blocks are slower under a malicious hint · repeated bad hints get peer-scored, like a badly-signed block

Second Temporal Window

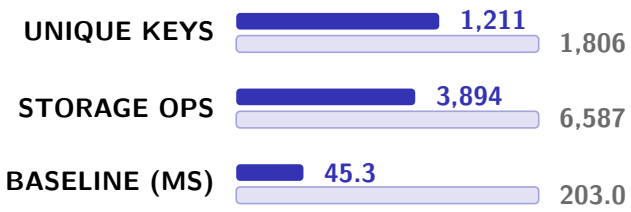
Bounded even under attack. Does any of this hold elsewhere?

1. 100,001 blocks, an earlier window
2. Lighter per-block workload
3. **10.1×** aggregate, 9.0× median speedup
4. **7.3%** hint overhead · 0.07% blocks slower

- **The mechanism holds as blocks get heavier**

MEDIAN WORKLOAD PER BLOCK

solid = earlier window · outline = primary window



WALL SPEEDUP, 16 THREADS



Same 16-thread prefetch setup, both windows

Generalizing Ira

It holds across two windows of Ethereum. Where else does it hold?

WHERE Ira APPLIES

One node executes first
it runs before its replicas

Replay is deterministic
same input, same access set

Replay is storage-bound
I/O is the bottleneck

CANDIDATE SYSTEMS

Replicated databases
CockroachDB, Calvin

Key-value stores
FoundationDB and kin

Blockchains
Solana, Sui, Aptos

No single leader needed: in DAG protocols (Narwhal, Bullshark) any validator that executed a batch can gossip its hints, and access sets from different proposers union.

When one node executes first, replicas should not have to rediscover what it already learned.

Paper and Artifacts



SCAN FOR THE PAPER

Paper

arxiv.org/abs/2601.21286

Code

github.com/adithyabhatkajake/ira-rs

tag v1.0-usenixsec26

Data

doi.org/10.5281/zenodo.20262361

hints · Parquet traces · state hashes

Questions?