

EthCache: A Workload-Aware, Unified Key-Value Cache for Ethereum

Jia Zhao[†], Yanjing Ren[†], Xianglong Shi[‡], Jingwei Li[‡], and Patrick P. C. Lee[†]

[†]The Chinese University of Hong Kong [‡]University of Electronic Science and Technology of China

Abstract—As Ethereum consensus grows more efficient, the performance bottleneck increasingly shifts to block processing, where transaction execution issues large volumes of key-value (KV) queries to persistent storage for account state, contract storage, bytecode, and chain metadata. Geth, the most widely used Ethereum client, reduces read latency with multiple in-memory caches, but these caches are scattered across semantic boundaries and the database layer, each governed by an independent LRU policy. Trace analysis of real Ethereum synchronization workloads reveals two properties that existing caches fail to exploit: 69.2% of unique KV pairs are read only once, causing severe LRU pollution, and transaction execution produces strong short-range co-read correlations that cross semantic boundaries. We present EthCache, a unified KV caching layer interposed between Geth’s execution engine and its storage backend. EthCache applies a dual-tier policy combining frequency-based promotion, timeout-based demotion, and recency-based eviction to suppress transient reads while retaining frequently read and correlated KV pairs in a shared capacity pool. Trace-driven evaluation shows that EthCache improves the cache hit rate over Geth’s default caches and sustains its advantage across dynamic workload phases and varying cache budgets.

I. INTRODUCTION

Blockchain platforms, such as Ethereum [6], provide decentralized execution and storage infrastructures for smart-contract applications [20], which have been used in domains including decentralized finance [27], supply chain management [23], and Internet of Things (IoT) [4]. However, public blockchains still provide limited transaction throughput [18]. This limitation stems not only from consensus protocols [10], [11], which govern transaction ordering and block production, but also from the storage layer, which affects transaction execution and state persistence. As consensus mechanisms become more efficient [26], storage I/O becomes an increasingly critical bottleneck.

In Ethereum clients, blockchain data (e.g., world state, historical block records, and auxiliary indexes) is persisted as key-value (KV) pairs in an underlying storage engine. A logical operation during transaction execution can translate into multiple KV reads, for example, when traversing trie nodes, fetching contract bytecode, or resolving metadata needed to validate a block. As the blockchain grows, these queries increasingly incur disk I/O, which can dominate block processing time and make data retrieval a critical factor in Ethereum performance [18], [25]. *Go-Ethereum (Geth)* [8] is the most widely used Ethereum execution client and relies on in-memory caches to reduce KV lookup latency.

However, Geth’s caches are designed as local optimizations rather than as a workload-aware storage layer. Geth scatters cache space across semantic-specific upper-layer caches and the database cache, all managed by simple Least Recently Used (LRU) policies. Thus, cache effectiveness depends on where a KV pair is accessed in the software stack, not on how useful it is to the current workload. A frequently reused state-trie node, a contract bytecode entry, and a block metadata entry may compete in separate cache spaces even when the same block-processing phase repeatedly accesses all three. Conversely, a burst of one-time reads can fill an isolated LRU cache and evict entries that would be reused shortly afterward.

Our workload analysis (§II-C) reveals two properties that explain why Geth’s cache organization is ineffective. First, KV reads are highly skewed: 69.2% of unique KV pairs are read only once, so every newly admitted entry has a high probability of never being reused before eviction, causing persistent LRU pollution. Second, transaction execution exhibits strong short-range read correlation due to data dependencies, such as traversing an account trie node followed immediately by its associated contract storage nodes and bytecode. These correlated reads frequently cross semantic boundaries, yet each of Geth’s isolated caches observes only its own subset of keys and cannot jointly retain correlated pairs. General-purpose cache policies, including classical (e.g., ARC [19] and LIRS [14]) and recently proposed ones (e.g., S3FIFO [30]), reduce cache pollution by exploiting recency, or frequency signals, yet none are designed for Ethereum’s specific combination of a massive read-once tail, strong short-range KV co-reads that cross semantic boundaries, and a scattered multi-cache architecture.

We propose EthCache, the first unified KV caching layer that co-designs cache unification with a workload-driven caching policy for Ethereum. EthCache sits between Geth’s transaction execution engine and its underlying KV storage engine. It replaces the scattered semantic caches with one shared KV cache and allocates memory based on observed reuse rather than fixed software boundaries. It adopts a *dual-tier* caching policy, which organizes cached entries into (i) a *probationary tier* for newly fetched and low-frequency KV pairs and (ii) a *stable tier* for entries that have accumulated sufficient read evidence. It applies (i) *frequency-based promotion* to keep one-time and short-lived reads in the probationary tier and move frequent reads to the stable tier, (ii) *timeout-based demotion* to give entries a second chance when their reuse has faded after a workload shift, and (iii) *recency-based eviction* to enforce

the global cache budget while favoring eviction from the probationary tier. This design retains correlated working sets without explicitly tracking all pairwise dependencies, which would be prohibitively costly under Ethereum’s long-tailed co-read distribution.

We evaluate EthCache with trace-driven replay against Geth’s default caches and representative replacement policies. EthCache improves the overall cache hit rate from 37.68% to 52.24% under the default cache size, and maintains its advantage across dynamic trace partitions and parameter settings. We open-source our EthCache prototype at <https://github.com/adslabcuhk/ethcache>.

We summarize the contributions as follows.

- **Workload characterization.** We analyze real Ethereum synchronization traces and identify two cache-relevant properties: a highly skewed read distribution and recurring short-range KV co-read correlations that cross semantic boundaries.
- **EthCache design.** We present a unified KV caching layer for Geth with a novel dual-tier policy.
- **Evaluation.** Trace-driven replay against Geth’s default caches and representative replacement policies shows that EthCache improves cache hit rates across various settings.
- **Availability.** The EthCache prototype is open-sourced.

II. BACKGROUND AND MOTIVATION

A. Blockchain Basics

A blockchain maintains a globally consistent, tamper-evident ledger via decentralized consensus [10], [11]. Each block contains a header with a cryptographic hash linking it to its predecessor and a body with a batch of digitally signed transactions. Consensus determines which blocks form the canonical chain, while each node independently re-executes those transactions to verify the resulting state transition.

Ethereum [6] extends this model with programmable transactions through *smart contracts* [20]. Executing a transaction involves more than appending a record: it may read account metadata, execute contract bytecode, access contract storage, and update the persistent state observed by future transactions. Ethereum represents this accumulated state as the *world state*, a globally shared, verifiable state machine tracking account balances, nonces, contract code, and contract storage. Nodes run client software to participate in and validate the network, with Geth serving as the predominant execution client.

B. Ethereum Storage and Caching

Figure 1 shows Geth’s data management and caching architecture.

Ethereum storage. Geth persists all Ethereum data as key-value (KV) pairs. Block headers and transaction bodies are stored as independent KV pairs, enabling retrieval by block number, block hash, or transaction identifier. The world state is maintained using Merkle Patricia Tries (MPTs), where every trie node is persisted as a single KV pair; a logical state lookup may require multiple sequential KV reads along a trie path before the requested account or storage slot is reached. Geth

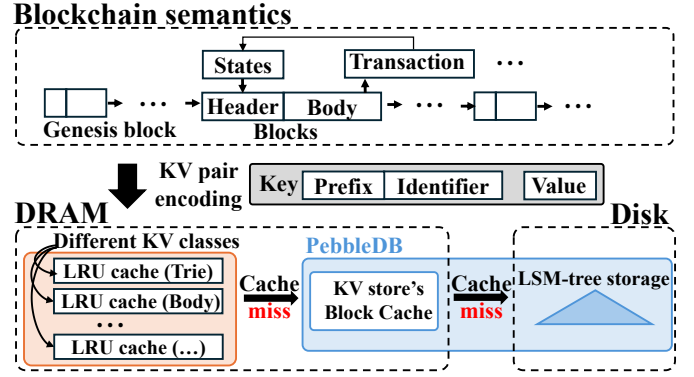


Figure 1: Geth’s storage architecture and caching hierarchy.

also stores auxiliary metadata, such as chain configurations and lookup indices, in the same database. To distinguish data types, Geth prepends a semantic-specific byte prefix to each key (e.g., ‘h’ for headers, ‘b’ for bodies) [9]; values are then serialized for compact storage. All KV pairs are written to an underlying Log-Structured Merge-tree (LSM-tree) engine, such as PebbleDB [16].

Geth’s caching architecture. To reduce direct KV-store read latency, Geth adopts a two-level caching hierarchy. The upper level consists of multiple independent LRU caches, each dedicated to a specific KV semantic. The lower level is PebbleDB’s built-in block cache, which stores recently accessed SSTable blocks while being agnostic to Ethereum-level key semantics.

The upper-level caches are configured as follows. The `--cache` flag (1 GB by default) allocates 50% to PebbleDB’s SSTable cache, 15% to the state read cache, 25% to the state dirty cache for buffered writes, and 10% to snapshot acceleration structures [28]. Outside this budget, Geth maintains a fixed 64 MB bytecode cache and four entry-limited caches: 512 entries each for block bodies and headers, 32 for execution receipts, and 1,024 for transaction lookup indices.

On a KV read, Geth first consults the upper-level cache matching the key’s semantic prefix. A hit returns the value immediately. A miss falls through to PebbleDB’s block cache; if the relevant SSTable block is resident, the read is served from DRAM. A second miss triggers a disk read, after which the fetched data may be inserted into one or more cache layers depending on its semantic type and the storage engine’s replacement policy. Consequently, a single execution-level request can interact with multiple independently managed cache spaces before reaching disk.

C. Limitation of Geth’s Caching Management

Workload analysis. To identify the root causes of caching inefficiency, we analyze KV operation traces from real Ethereum synchronization workloads (see §V-A for trace details), studying both individual read frequency and short-range co-reads. An effective cache must retain not only frequently read individual KV pairs but also groups of KV pairs that recur together during transaction execution.

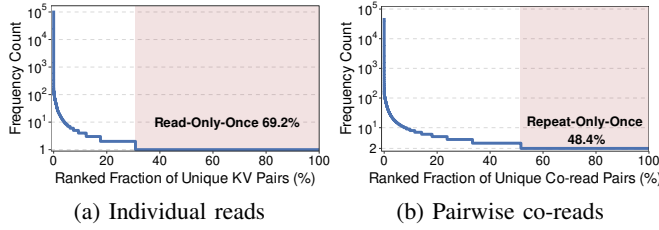


Figure 2: Frequency distributions of individual reads and pairwise co-reads.

Figure 2 shows the frequency distributions of individual KV reads and pairwise co-reads; a *co-read* is defined as an unordered adjacent read pair, capturing short-range locality without requiring knowledge of higher-level transaction semantics. Co-read pairs are counted only if they recur at least twice across the full trace.

- *Skewed read frequency with a massive read-once tail.* Figure 2(a) shows that KV-pair reads follow a highly skewed frequency distribution, where a small hot set is accessed repeatedly while most KV pairs are read infrequently. The tail is significant, with 69.2% of unique KV pairs read only once. This pattern is challenging for a pure recency policy because one-time reads still consume cache space when they are admitted, even though many of them will not be reused before eviction.
- *Strong short-range read correlation with a long-tailed co-read distribution.* Correlated reads are common at short distances, as evidenced by the massive number of adjacent co-reads in Figure 2(b) and corroborated by prior work [25]. This strong short-range correlation arises naturally from the data dependencies in Ethereum workloads. For example, reading an account node from the state trie often triggers subsequent accesses to its associated contract storage nodes and bytecode. Also, the frequency distribution of co-reads is highly skewed and long-tailed: 48.4% of co-read pairs repeat only once. The reusable part of the co-read distribution indicates that some KV pairs should be retained together, while the long tail indicates that explicitly tracking all pairwise relationships would add substantial metadata for little benefit.

Limitations. The above observations expose three limitations in Geth’s current caching management.

First, scattered cache spaces can lead to redundant memory consumption, as the same logical data may be cached repeatedly in both the upper-layer semantic caches and lower-layer SSTable cache. Because these layers are managed independently, memory assigned to one cache cannot be reused by another cache even when the latter faces stronger reuse.

Second, Geth applies simple LRU eviction within isolated caches without incorporating the workload skewness or correlations revealed by the trace analysis. A KV pair that is frequently co-read with entries from another semantic cache cannot benefit from their combined read pattern, because each cache observes only its own key subset. This isolation makes it difficult for the cache hierarchy to retain correlated working sets across semantic boundaries.

Third, the pronounced long-tail read distribution induces

severe cache pollution under simple LRU, since the large volume of low-frequency reads can rapidly evict correlated, high-frequency KV pairs. Recency alone cannot distinguish a newly fetched one-time trie node from a KV pair that repeatedly appears in transaction execution, so transient reads may temporarily look as valuable as stable working-set entries.

Taken together, Geth’s scattered caches cannot distinguish transient low-frequency reads from genuine working sets, and they cannot allocate capacity according to observed reuse across KV semantics. These three limitations motivate a unified caching layer that manages all KV pairs in one shared space, filters the read-once tail, and protects high-frequency entries from long-tail interference.

III. DESIGN

A. Design Overview

Design considerations. The trace analysis (§II-C) shows that Ethereum KV caching should be driven primarily by read frequency, as most cache pollution originates from KV pairs touched only once or a few times during transaction execution. The key question is how much frequency information is necessary. Since most KV pairs belong to the transient tail while a small subset is read repeatedly, the cache primarily needs to distinguish short-lived reads from the stable working set; finer-grained frequency levels yield diminishing benefit beyond this binary distinction while inflating per-entry metadata and pointer-movement overhead. The co-read distribution follows the same skewed pattern, so KV pairs that are frequently read together recur in similar transaction-processing contexts and can be retained jointly by a frequency-aware policy without explicit dependency tracking. To this end, EthCache uses a single frequency threshold to identify KV pairs for stable residency, and combines this threshold with lightweight aging to reclaim entries that were once active but have since become idle.

Unified caching architecture. EthCache replaces Geth’s scattered semantic caches with a unified cache placed between the execution engine and the storage backend. Rather than dividing memory across separate cache layers and semantic-specific upper caches, EthCache lets all KV pairs share a single physical cache space and compete under the same policy, so capacity is allocated based on observed workload behavior rather than on fixed software boundaries.

The unified cache comprises two components: (i) a *hash map* physically stores cached KV pairs and serves lookups on cache hits, and (ii) a *policy engine* tracks the read frequency and recency of each cached entry and decides when an entry should remain probationary, become stable, or be evicted. The policy engine is split into two logical tiers: (i) a *probationary tier* that absorbs newly admitted and infrequently read KV pairs, and (ii) a *stable tier* that retains KV pairs whose read frequency has crossed a configured promotion threshold. The two tiers share the same global cache budget; entries migrate between them according to workload behavior, so the tier boundary guides replacement decisions without statically partitioning cache space.

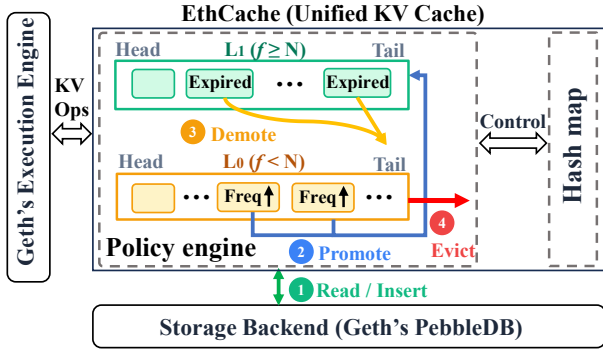


Figure 3: EthCache’s design.

B. Design Details

Figure 3 illustrates EthCache’s components and workflows.

Dual-tier organization. Since Ethereum reads are dominated by a long read-once tail with a small repeatedly read core (§II-C), EthCache only needs to separate transient reads from the stable working set rather than maintain fine-grained hotness levels that would inflate metadata and pointer-movement cost. It organizes policy metadata into two LRU-ordered tiers separated by a single promotion boundary: L_0 (probationary tier) cheaply tracks newly admitted entries, whereas L_1 (stable tier) protects repeatedly read entries that demonstrate persistent reuse. Each policy entry maintains a frequency counter f for promotion and a last-access counter t for aging. A global KV-operation counter T_g starts from 0 and increments by one on each KV operation; t is updated to T_g whenever the entry is read or written. Note that the KV pairs themselves remain physically stored in the hash map.

Tiered management. EthCache manages the two tiers with *frequency-based promotion*. It introduces a *promotion threshold* N : an entry is tracked in L_0 while $f < N$ and is promoted to the tail of L_1 once $f \geq N$, where $N = 3$ by default. A KV pair must thus be observed across at least N reads before it can compete for inclusion in the stable working set. Frequency-based promotion also captures correlated KV pairs without explicitly detecting pairwise correlations. Such KV pairs recur in similar transaction-processing contexts (§II-C), so registering newly fetched entries in L_0 lets correlated pairs accumulate read frequency across repeated contexts and enter L_1 , while one-off reads remain in L_0 and are quickly displaced. Thus, EthCache requires neither pairwise-correlation metadata nor prior knowledge of transaction-level dependencies.

EthCache uses *timeout-based demotion* to move stale entries out of the stable tier L_1 to the probationary tier L_0 , instead of evicting them immediately. It proactively checks whether the coldest stable-tier entry has become stale. On each cache operation, it inspects the LRU tail of L_1 and computes the idle time as $T_g - t$. If the idle time exceeds an *expiration threshold* E , EthCache removes the entry from L_1 , resets its frequency counter f to 0, and moves it to the LRU tail of L_0 without deleting the corresponding KV pair from the hash map. This demotion gives the entry a second chance before eviction. This cleanup prevents once-hot KV pairs from occupying L_1

indefinitely after the workload shifts. A small E aggressively reclaims stable-tier space at the cost of prematurely demoting entries that may still be reused, whereas a large E retains more stable entries but slows the cleanup of stale ones; §V examines this trade-off empirically.

EthCache uses *recency-based eviction* only when an insertion or update would exceed the global cache budget. Within each tier, LRU ordering selects eviction candidates by recency, so policy entries for low-frequency KV pairs in L_0 are quickly displaced under memory pressure, while frequently read and co-read KV pairs accumulate enough reads to enter L_1 . The policy engine selects the victim from the LRU tail of L_0 and removes it from both the policy engine and the hash map. This priority keeps L_0 as a high-turnover region for the read-once tail and prevents transient KV pairs from displacing stable entries prematurely. If L_0 is empty, EthCache falls back to the LRU tail of L_1 and evicts that stable-tier entry instead, preserving a single global cache budget even when most resident entries have been promoted.

Cache read/write. Cache reads and writes update both the hash map and the corresponding policy entries. As described above, new entries created by read misses or writes are admitted to L_0 , so that EthCache can capture both read reuse and read-after-write locality without letting writes alone populate L_1 .

On a read, EthCache first looks up the requested KV pair in the hash map. A cache miss fetches the KV pair from the underlying KV store, inserts it into the hash map, and registers its policy entry in L_0 . If this insertion exceeds the cache budget, EthCache applies recency-based eviction. A cache hit updates the entry’s LRU position, frequency counter f , and last-access counter t ; once f reaches the promotion threshold N , EthCache moves the policy entry to L_1 as part of the stable working set.

On a write, EthCache inserts or updates the KV pair in the hash map. An uncached KV pair receives a new policy entry in L_0 . For an entry already in L_0 , the policy engine refreshes its recency position and last-access counter t but does not increase its frequency counter, so write bursts alone cannot promote rarely read KV pairs to L_1 . For an entry already in L_1 , the policy engine refreshes its recency position and last-access counter t without resetting its frequency counter, allowing the entry to retain its stable-tier membership. If the insertion or value update exceeds the cache budget, EthCache applies recency-based eviction before updating the written KV pair. Thus, frequently read KV pairs remain stable across occasional updates, while newly written KV pairs must accumulate read evidence before entering L_1 .

IV. IMPLEMENTATION

We implement EthCache in Go to compare the proposed design against Geth’s default caching model and representative cache policies, including LIRS [14], ARC [19], S3-FIFO [30]. We replay KV operations from the real-world Ethereum workload (§V-A) in their original order. For each read, we check whether the requested KV pair is cached and update the cache state according to the evaluated policy. On a miss, the KV pair is admitted according to the policy’s admission and

retention rules. For each write, we update the cache according to the corresponding write-allocation behavior, which may insert or update the written KV pair and evict existing entries if needed. EthCache is a read cache, so the reported hit rate is computed only from read operations; write operations are replayed to preserve the original workload order and to update cache contents under write allocation, since they can influence subsequent read hits.

Geth modeling. We faithfully implement Geth’s upper-layer caches by identifying the semantics of each KV pair from its key prefix [9] and maintaining an independent LRU cache per semantic category (§II-B). For the lower layer, we model PebbleDB’s block cache as a logical KV-level LRU cache rather than reproducing its SSTable-block structure. This modeling is necessary since a reported SSTable cache “hit” in PebbleDB can be misleading: the hit may correspond to an index or filter block rather than the requested KV pair, in which case a disk read is still needed to retrieve the actual data. Modeling the lower-level cache at the KV granularity eliminates this ambiguity and directly measures the true data-retrieval hit rate.

This KV-level abstraction is deliberately conservative with respect to Geth: it omits the block-level indexing and filter overhead, along with the read amplification, inherent to a real LSM-tree block cache, thereby yielding an optimistic hit-rate estimate for the baseline. Thus, the improvements of EthCache over the idealized Geth model represent a lower bound on its real-world advantage; the gain over an actual Geth/PebbleDB deployment would be at least as large, since the unmodeled overheads would further reduce Geth’s effective hit rate without similarly affecting EthCache.

V. EVALUATION

We evaluate EthCache through trace-driven replay. We first compare its hit rate against several baselines under the same cache budgets (Exp#1): Geth’s default caching model (which retains Geth’s original scattered, semantic-specific LRU caches described in §II-B), Unified LRU (which merges all KV pairs into a single shared cache space under plain LRU replacement, isolating the effect of unification alone from EthCache’s dual-tier policy), and three additional replacement policies: S3FIFO [30], ARC [19], and LIRS [14], each implemented over the same unified cache space as Unified LRU but governed by its own eviction and admission logic in place of plain LRU. We then translate hit-rate gains into expected read-latency reductions (Exp#2), evaluate robustness across trace partitions (Exp#3), and verify whether EthCache’s hits come from the correlated KV accesses identified in §II-C (Exp#4). Finally, we quantify the metadata overhead (Exp#5) and evaluate the sensitivity to the promotion threshold N (Exp#6) and the expiration threshold E (Exp#7). Except for Exp#5, cache budgets follow Geth’s convention and count only cached KV payloads (key and value); Exp#5 charges metadata to the same total memory budget as cached payloads to evaluate whether EthCache maintains its advantage under a more realistic accounting.

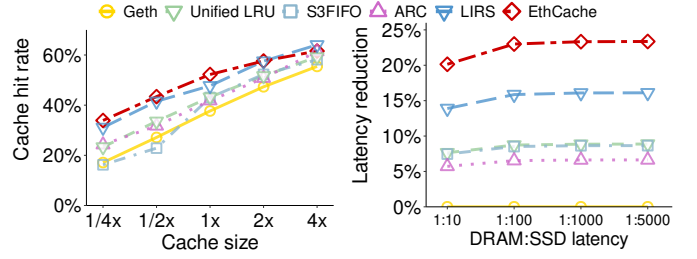


Figure 4: Exp#1: Cache hit rates. Figure 5: Exp#2: Average read latency reduction over Geth.

A. Methodology

Dataset. We collect KV operation traces from Ethereum’s storage backend using a modified Geth client. We disable Geth’s LRU caches during collection to capture raw KV accesses without caching interference, while retaining snapshot acceleration to preserve realistic synchronization behavior. Each KV operation records its timestamp, key, and value size, allowing the replay engine to apply consistent capacity accounting across policies despite variable entry sizes.

We focus on the 50 K blocks following the synchronization of 20.5 M blocks to reflect steady-state synchronization. This trace contains 262.5 M KV operations, comprising 61.1 M reads and 201.4 M writes, over a persistent storage footprint of 20.1 GB. Since EthCache is a read cache, our primary metric is the *read hit rate*, defined as the fraction of KV read operations served from cache during replay. Writes in Geth are buffered and latency-insensitive on the critical path, but we still replay them as they update cache contents and can affect later read hits.

Testbed. All experiments run on a server with an Intel(R) Xeon(R) Silver 4215 CPU, 512 GB of RAM, and a 3.5 TB NVMe SSD, running Ubuntu 24.04.4 LTS with Linux kernel 6.17. Unless otherwise stated, EthCache uses $N = 3$ and $E = 10M$ operations (approximately 1,900 blocks at this trace’s average rate of around 5,250 operations per block); we justify the defaults empirically in Exp#6 and Exp#7.

B. Results

Exp#1 (Cache hit rates). We compare EthCache, Geth’s caches, unified LRU, S3FIFO, ARC, and LIRS across different cache sizes. Geth’s default read-cache configuration is the $1\times$ setting, whose caches (§II-B) total 778.2 MB. We scale the cache limits to obtain $1/4\times$, $1/2\times$, $2\times$, and $4\times$ settings.

Figure 4 shows the results. EthCache achieves the highest hit rate from $1/4\times$ to $2\times$ and remains competitive at $4\times$. At the default cache size, EthCache reaches 52.24%, improving over Geth’s 37.68% by 14.56 percentage points (pp). This gain comes from a unified capacity pool across KV semantics and a dual-tier policy that keeps the read-once tail out of stable cache space.

Unified LRU, S3FIFO, ARC, and LIRS improve over Geth, but still trail EthCache by 9.04 pp, 9.17 pp, 10.43 pp, and 4.52 pp, respectively. S3FIFO filters one-time reads, but its FIFO replacement does not protect repeatedly useful pairs as they age. ARC adapts its capacity split, but its permissive promotion admits transient burst reads into the frequent list.

LIRS accounts for reuse distance yet still admits short-lived repeated reads with low inter-reference recency. EthCache’s tunable threshold and timeout-based expiration filter short-lived reuse while aging out stale entries.

As the cache size grows, all hit rates rise because more of the synchronization working set remains resident. At $4\times$, the cache holds around 15% of persistent storage, where EthCache reaches 61.58%, 6.13 pp above Geth (55.45%) and close to LIRS (64.04%). EthCache’s relative advantage over LIRS narrows as the cache size grows, which suggests that its admission and demotion mechanisms matter most precisely in the memory-constrained regimes typical of production Geth deployments, and less so when capacity is abundant enough that most working-set entries fit regardless of policy.

Exp#2 (Expected performance under DRAM-to-SSD hierarchy). We quantify how the hit-rate gains translate into expected read latency under a DRAM-to-SSD hierarchy. We use the *average access time (AAT)* model [12], $AAT = HitTime + MissRate \times MissPenalty$, where the miss penalty denotes the SSD access cost normalized by the DRAM hit time. Since this ratio depends on the storage device and platform, we consider four miss-penalty settings, 1:10, 1:100, 1:1000, and 1:5000.

Figure 5 reports each approach’s average read latency reduction over Geth at the default $1\times$ cache size. EthCache achieves the largest latency reduction under all miss-penalty settings. With a 1:1000 DRAM-to-SSD ratio [12], EthCache reduces average read latency by 23.33% over Geth, and its reduction over Geth is $2.64\times$, $2.70\times$, $3.52\times$, and $1.45\times$ compared to those of unified LRU, S3FIFO, ARC, and LIRS, respectively. EthCache’s reduction increases from 20.14% at 1:10 to 23.00% at 1:100, and then changes only slightly to 23.33% and 23.36% at 1:1000 and 1:5000. This saturation is expected from the AAT model, since the hit-time term becomes negligible once SSD misses dominate and the normalized latency reduction is governed mainly by miss-rate reduction. Thus, EthCache’s hit-rate advantage in Exp#1 translates into a stable latency advantage across both modest and aggressive DRAM-to-SSD latency gaps.

Exp#3 (Cache efficiency under dynamic workloads). Since Ethereum synchronization is not stationary and different block ranges can involve different contracts and accounts, a policy that performs well only after a long warmup or only on a single interval may not improve real synchronization. We examine whether EthCache remains effective as the workload changes.

Figure 6(a) reports the cumulative hit rate over the full 50K-block trace, which captures cache warmup and long-term retention. EthCache maintains the highest cumulative hit rate throughout the trace. Its hit rate increases from 46.49% at the first 10K-trace point to 52.24% at the end, while Geth remains within 36–38% and ends at 37.68%. Unified LRU, S3FIFO, ARC, and LIRS converge to 43.20%, 43.07%, 41.81%, and 47.72%, respectively. This trend suggests that the benefit of EthCache accumulates over time. The stable tier retains KV pairs that repeatedly reappear across block-processing phases, while timeout-based demotion prevents entries useful in earlier

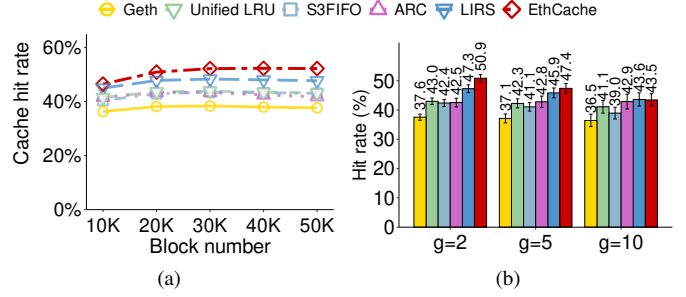


Figure 6: Exp#3: Cache efficiency under dynamic workloads. (a) Cache hit rates over time across the whole trace; (b) Average hit rates across trace partitions, with error bars showing standard deviations.

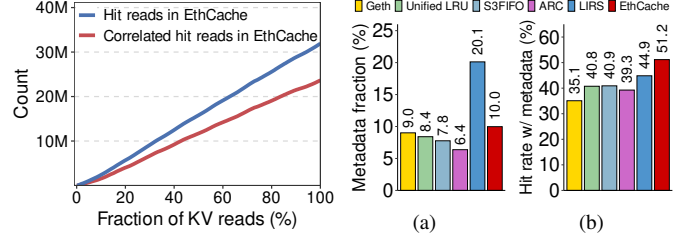


Figure 7: Exp#4: Correlation analysis of KV pairs captured by the cache.

Figure 8: Exp#5: Metadata analysis. (a) Metadata overhead. (b) Hit rate with metadata.

phases from permanently consuming protected space after the workload shifts.

Figure 6(b) varies the evaluation granularity by partitioning the trace into g consecutive groups, where $g \in \{2, 5, 10\}$; for each g , we replay every sub-trace independently with a cleared cache and report the average hit rate, with error bars showing the standard deviation. We see that EthCache’s advantage is not an artifact of one trace partition. It achieves the highest average hit rate for all three partition choices. When $g=5$, it reaches 47.4%, outperforming Geth by 10.3 pp and LIRS by 1.5 pp. The same ordering persists when the trace is split more coarsely ($g=2$) or more finely ($g=10$), and the standard-deviation error bars remain small relative to the mean gaps. Replaying each sub-trace with a cleared cache makes this test stricter than the cumulative curve, because every partition must rebuild its working set from scratch. These results show that EthCache is robust across both long and short phases of Ethereum’s KV read stream, in which its promotion threshold filters transient reads while its timeout-based demotion adapts to phase changes.

Exp#4 (Correlation analysis). We measure whether EthCache’s hits come from correlated KV pairs, which are the short-range co-read patterns identified in §II-C. We first extract recurring co-read pairs from the raw read trace using the same definition as the workload analysis, namely unordered adjacent read pairs that appear at least twice. During trace replay, we mark a cache hit as a correlated hit only when its adjacent partner belongs to one of these recurring co-read pairs and both reads in the pair hit in EthCache. Such a pair contributes two correlated hit reads, one for each KV read. We then compare the cumulative number of correlated hit reads with the cumulative number of all hit reads in EthCache.

Figure 7 shows that correlated KV pairs account for most of EthCache’s cache hits. At the end of the trace, EthCache serves 31.9M hit reads, among which 23.7M are correlated hit reads, accounting for 74.3% of all hits. This high fraction indicates that EthCache does not merely retain isolated hot KV pairs; it also preserves KV pairs that repeatedly appear together during transaction execution. This result also rules out a simpler explanation of the hit-rate gain. If EthCache only benefited from retaining a few individually hot keys, correlated hits would form a much smaller fraction of total hits. Instead, the measured fraction supports the design insight in §III-A: frequency-based promotion can capture correlated working sets without explicitly tracking all pairwise dependencies, while timeout-based demotion removes correlated pairs once their transaction-processing context becomes stale.

Exp#5 (Metadata analysis). We evaluate whether policy metadata affects the hit-rate comparison under the default $1\times$ memory budget. For each policy, we count metadata separately from cached KV payloads, including pointers, counters, size fields, ghost entries, queue state, and estimated Go map runtime overhead. We then charge the metadata to the same total memory budget and replay the trace with the remaining capacity available for KV payloads.

Figure 8(a) shows that metadata overhead is policy-dependent. ARC has the smallest footprint, at 6.4% of cached-payload size, followed by S3FIFO at 7.8%, unified LRU at 8.4%, and Geth at 9.0%. EthCache incurs 10.0% metadata overhead because it stores the frequency, recency, and timestamp state for dual-tier cache management. LIRS has the highest overhead, 20.1%, as it tracks both resident and non-resident entries and maintains multiple stack and queue links per entry.

Figure 8(b) shows that EthCache remains the most effective after metadata is charged to the cache budget. Relative to the original $1\times$ hit rates in Figure 4, EthCache decreases from 52.24% to 51.17% (i.e., 1.07 pp loss). The baselines lose more capacity to metadata, with Geth decreasing from 37.68% to 35.09%, unified LRU from 43.20% to 40.75%, S3FIFO from 43.07% to 40.91%, ARC from 41.81% to 39.25%, and LIRS from 47.72% to 44.85%. EthCache still preserves its hit-rate advantage with consideration of metadata overhead because its admission and demotion policy outweighs the modest cost of its additional metadata.

Exp#6 (Impact of promotion threshold). We evaluate the impact of the promotion threshold N by varying it from 1 to 6. A smaller N admits entries earlier, while a larger N filters transient reads more aggressively but may delay useful promotions. Figure 9 shows that $N = 3$ achieves the highest hit rate, 52.24%, improving over Geth by 14.56 pp. Setting N too small weakens the filter, reducing the hit rate to 41.90% at $N = 1$ and 42.58% at $N = 2$ as short-lived repeated reads enter the stable tier and displace longer-lived entries. In contrast, setting N too large delays promotion of reusable KV pairs, reducing the hit rate to 51.47%, 49.90%, and 49.19% for $N = 4$, $N = 5$, and $N = 6$, respectively. The larger drop at small N indicates that premature promotion is more harmful than

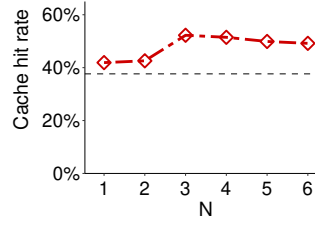


Figure 9: Exp#6: Impact of promotion threshold N compared with Geth.

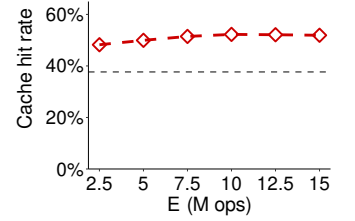


Figure 10: Exp#7: Impact of expiration threshold E compared with Geth.

delayed promotion on this trace, which also suggests why ARC’s permissive promotion underperforms on this workload. Thus, $N = 3$ strikes the best balance between filtering transient reuse and admitting genuinely reusable KV pairs; dynamically adapting N across synchronization phases is future work.

Exp#7 (Impact of expiration threshold). We evaluate the impact of the expiration threshold E by varying it from 2.5M to 15M operations; E controls how long a stable-tier entry can remain idle before demotion, trading off reuse preservation and stale-entry cleanup. Figure 10 shows that EthCache is effective across all timeout values, with hit rates from 48.23% to 52.24%, all at least 10.55 pp above Geth. The best setting is $E = 10$ M, which reaches 52.24% and improves over Geth by 14.56 pp. Shorter timeouts demote useful stable entries too early, reducing the hit rate to 48.23% and 49.94% at 2.5M and 5M operations, while longer timeouts retain stale KV pairs longer and reduce the hit rate to 52.11% and 51.90% at 12.5M and 15M operations. The curve is relatively flat once E reaches 7.5M operations, so the timeout should span common reuse intervals but does not require fine-grained tuning.

VI. RELATED WORK

Ethereum I/O optimizations. Ethereum’s storage stack has evolved along several directions to reduce I/O cost and storage growth. At the client level, Geth moves immutable historical blocks and receipts into the freezer database [7], stores trie nodes by path for in-place updates in PathDB [21], and bounds long-term storage via historical data limits and pruning [15]. However, these mechanisms do not optimize read-cache behavior on the critical path of block processing.

Some studies redesign authenticated data structures (ADSs) or their interaction with storage: RainBlock [22] decouples execution from storage I/O with a multi-version ADS; LMPTs [3] cache recent updates in layered tries; LVMT [17] adopts a versioned multipoint trie; COLE [31] stores historical states columnarily with learned indexes; LETUS [29] combines authenticated indexing with log-structured storage.

Some studies optimize the underlying KV store: mLSM [24] organizes tries across hierarchical levels; Block-LSM [2] groups updates by block number; ChainKV [1] separates data by access semantics; MoltDB [18] segregates ancient states during compaction. In contrast, EthCache is a protocol-transparent caching layer between transaction execution and the KV store, reducing read latency by exploiting Ethereum-specific read patterns without modifying the ADS, proof format,

or physical storage layout.

Caching policies. General-purpose cache policies improve over LRU by exploiting recency, frequency, or reuse distance, but they do not directly target Ethereum’s combination of skewed reads, short-range KV correlations, and a large low-frequency tail. ARC [19] uses ghost entries to balance recency and frequency, while LIRS [14] and CLOCK-Pro [13] protect entries with low inter-reference recency. TinyLFU [5] admits items by approximate frequency, but its per-item admission does not capture the short read sequences produced by Ethereum transactions. S3FIFO [30] filters one-time reads with multiple FIFO queues, yet its eviction order remains tied to insertion time. In contrast, EthCache combines frequency-based filtering with recency-based cleanup to preserve frequently read and correlated KV pairs while filtering transient accesses.

VII. CONCLUSION AND FUTURE WORK

Ethereum clients increasingly face storage I/O bottlenecks during block processing, as transaction execution and state updates issue large volumes of KV queries to persistent storage. Existing Geth caches mitigate this cost only partially, since their scattered cache spaces and independent LRU policies do not match Ethereum workloads with skewed reads, transient low-frequency items, and short-range correlations across KV pairs. This paper presents EthCache, a unified KV caching layer that manages KV pairs of different semantics in a shared cache space and combines frequency-based promotion, recency-based eviction, and timeout-based demotion to filter transient reads while preserving frequently read and correlated KV pairs. Trace-driven evaluation shows that EthCache substantially improves cache effectiveness and accelerates block processing over Geth. In future work, we plan to integrate EthCache into Geth for end-to-end evaluation.

Acknowledgments. This work was supported in part by the National Natural Science Foundation of China (62522204 and 62561160158), Open Research Fund of State Key Laboratory of Internet Architecture under Grant (HLW2025ZD04), and NSFC/RGC Joint Research Scheme (N_CUHK403/25).

REFERENCES

- [1] Z. Chen, B. Li, X. Cai, Z. Jia, L. Ju, Z. Shao, and Z. Shen. ChainKV: A semantics-aware Key-Value store for Ethereum system. *Proceedings of the ACM on Management of Data*, 1(4):1–23, 2023.
- [2] Z. Chen, B. Li, X. Cai, Z. Jia, Z. Shen, Y. Wang, and Z. Shao. Block-LSM: An ether-aware block-ordered LSM-tree based key-value storage engine. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 25–32, 2021.
- [3] J. A. Choi, S. M. Beillahi, P. Li, A. Veneris, and F. Long. LMPTs: Eliminating storage bottlenecks for processing blockchain transactions. In *Proceedings of IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2022.
- [4] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram. Blockchain for IoT security and privacy: The case study of a smart home. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 618–623, 2017.
- [5] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A highly efficient cache admission policy. *ACM Transactions on Storage*, 13(4):1–31, 2017.
- [6] Ethereum Foundation. Ethereum. <https://ethereum.org/>, 2026.
- [7] Ethereum Foundation. go-ethereum freezer database. <https://geth.ethereum.org/docs/fundamentals/databases>, 2026.
- [8] Ethereum Foundation. go-ethereum (Geth). <https://geth.ethereum.org>, 2026.
- [9] Ethereum Foundation. go-ethereum’s KV storage schema. <https://github.com/ethereum/go-ethereum/blob/master/core/rawdb/schema.go>, 2026.
- [10] Ethereum Foundation. Proof of stake. <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/>, 2026.
- [11] Ethereum Foundation. Proof of work. <https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/>, 2026.
- [12] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [13] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 323–336, 2005.
- [14] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 31–42, 2002.
- [15] G. Kadianakis, lightclient, and A. Stokes. EIP-4444: Bound historical data in execution clients prune historical data in clients older than one year. <https://eips.ethereum.org/EIPS/eip-4444>, 2021.
- [16] C. Labs. Pebble. <https://github.com/cockroachdb/pebble>, 2025.
- [17] C. Li, S. M. Beillahi, G. Yang, M. Wu, W. Xu, and F. Long. LVMT: An efficient authenticated storage for blockchain. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–153, 2023.
- [18] J. Liang, W. Chen, Z. Hong, H. Zhu, W. Qiu, and Z. Zheng. MoltDB: Accelerating blockchain via ancient state segregation. *IEEE Transactions on Parallel and Distributed Systems*, 2024.
- [19] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2003.
- [20] B. K. Mohanta, S. S. Panda, and D. Jena. An overview of smart contract and use cases in blockchain technology. In *Proceedings of IEEE International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018.
- [21] NodeReal. Geth path-based storage model and newly inline state prune. <https://nodereal.io/blog/en/geth-path-based-storage-model-and-newly-inline-state-prune/>, 2023.
- [22] S. Ponnappalli, A. Shah, S. Banerjee, D. Malkhi, A. Tai, V. Chidambaram, and M. Wei. RainBlock: Faster transaction processing in public blockchains. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 333–347, 2021.
- [23] M. M. Queiroz, R. Telles, and S. H. Bonilla. Blockchain and supply chain management integration: a systematic review of the literature. *Supply Chain Management: An International Journal*, 25(2):241–254, 2020.
- [24] P. Raju, S. Ponnappalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham. mLSM: Making authenticated storage faster in Ethereum. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.
- [25] Y. Ren, J. Zhao, J. Li, and P. P. C. Lee. An analysis of Ethereum workloads from a Key-Value storage perspective. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, pages 394–406, 2025.
- [26] F. Saleh. Blockchain without waste: Proof-of-stake. *The Review of Financial Studies*, 34(3):1156–1190, 2021.
- [27] F. Schär. Decentralized finance: on blockchain and smart contract-based financial markets. *Review of the Federal Reserve Bank of St. Louis*, 103(2):153–174, 2021.
- [28] P. Szilágyi. Ask about Geth: Snapshot acceleration. <https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration>, 2020.
- [29] S. Tian, Z. Lu, H. Zhuo, X. Tang, P. Hong, S. Chen, D. Yang, Y. Yan, Z. Jiang, H. Zhang, et al. LETUS: A log-structured efficient trusted universal blockchain storage. In *Companion of ACM International Conference on Management of Data (SIGMOD)*, pages 161–174, 2024.
- [30] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 130–149, 2023.
- [31] C. Zhang, C. Xu, H. Hu, and J. Xu. COLE: A column-based learned storage for blockchain systems. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 329–345, 2024.