Furrballs: Topology-Aware, High-Performance Resource Placement for Asymmetric Memory Systems

Technical Report v1.34.0

Hamza Jamil Saied (The Sphynx)

2026-06-13

Table of contents

Status

v1.34.0 — ATC 2026 paper revisions based on simulated reviewer feedback. Fixed citations (Berg et al. OSDI 2020 for CacheLib, TPP, Nimble, AutoTiering). Corrected working-set math logic for oversubscription. Clarified YCSB-C latency anomaly as a load-order artifact. Expanded paper with Staging Pages and CacheLibNuma comparison sections, and provided a concrete CXL example. Paper length expanded to 5 pages. Ready for next EC2 session for adversarial workload.

v1.33.0 — 4T anomaly root cause identified and fixed. `alignas(64)` statistics atomics on the FurrBall heap object caused cross-socket cache-line bouncing on 4+ NUMA nodes: every GET touched 3 cache lines on node 0 from remote sockets, adding ~360ns at Infinity Fabric distance 32. Fix: per-node stat sharding (hot-path counters moved into NUMA-allocated `PerNodeDetails`). Result: 4T p50 GET 587ns -> 71ns (-88%), 2T p50 GET 120ns -> 74ns (-39%). Full NUMABench + YCSB suite re-run on c6a.metal with fix applied. YCSB workloads A/B/C validated across 5 systems at 64B/256B/1024B. **v1.32.0** — Cross-platform validation on AMD EPYC Milan (c6a.metal, 4 NUMA nodes, Infinity Fabric). Topology-aware placement confirmed on a second

vendor and topology: FurrBallTL 1.5x faster GET than CacheLib at 64B, 9.5x faster at 128MB working-set (UniformRO 1024B). CacheLibNuma = CacheLib on AMD (per-pool routing without mbind is ineffective). Lock-distribution signal amplified on AMD (TL vs SN: 3.4x on Milan vs 2.0x on Ice Lake). 4T TL anomaly (6.6x GET degradation from 2T) identified but unexplained — root cause deferred. 212 benchmarks across 12 adapters on c6a.metal, zero crashes. v1.31.0: Staging pages eliminate the SET tail-latency bottleneck for ARC/LRU eviction policies. The SET path now has a guaranteed O(1) fast path (bump or staging), decoupling SET from space management. EC2 validation on c6i.metal (Intel Xeon Platinum 8375C): p99_set improved 24–63x (68K to 2.8K ns at 64B, 200K to 4.5K ns at 1024B) with zero GET regression. Page drain compaction (proactive maintenance) creates a virtuous cycle: eviction empties sparse pages, drain consolidates remaining keys, recycled pages become fresh bump targets. Destructor race condition fixed (refcount + double-check replaces heuristic 20ms sleep). CMap h2 truncation bug fixed (KeyH2 vector now stores full 64-bit hash, not truncated 32-bit). FurrBallSN vs CacheLib at equal usable capacity: 10.9x (64MB) and 13.9x (128MB) lower p50 GET latency, 100% vs 50% memory efficiency.

Abstract

Furrballs is a topology-aware resource placement library for asymmetric memory systems. Asymmetric memory — where memory domains have different access latencies (NUMA, CXL-attached memory, HBM tiers) — is the defining characteristic of modern server memory hierarchies. Furrballs uses memory topology as a first-class input to per-page placement and eviction decisions. The core contribution is per-page allocation on specific memory domains, with ARC (Adaptive Replacement Cache) for eviction and SimpleMigratePolicy for tier-aware migration. SimpleMigratePolicy uses two 4-bit saturating counters per key (local and remote access counts) with exponential decay, providing migration decisions without the overhead of REMARC's framework (lookup tables, SIMD scanning, algebraic composition). Pages are pre-allocated at initialization via one physical allocation per memory domain (NUMA node in the current implementation), subdivided into logical pages filled by a bump allocator. Key routing supports round-robin distribution and thread-local placement (via a runtime config flag). The library is implemented in C++20, uses a platform abstraction layer (Numatic) for memory topology operations, and integrates a custom synchronization library (StreamLine) for thread coordination.

A companion paper [Sphynx2025] (DOI: 10.5281/zenodo.19794758) documents the REMARC (Reduction-Modeled Adaptive Replacement Cache) framework that was originally developed for this system. After extensive evaluation — including single-node eviction (score degeneracy, §10), multi-node placement simulation (K=2, K=3, predictive strategies, feedback dimensionality, §11), and theoretical analysis (cold-entry information hole, feedback SNR) — REMARC was found to add zero value over simple EMA counting. The framework is retained as a thorough negative finding. The system now uses SimpleMigratePolicy as the production path and ARC for eviction.

Phase 1 validated the core architecture using lock-free reads via SeqLock on per-node sharded KeyMeta. Phase 2a replaces the Phase 1 data structures with CMap, a lock-free-read concurrent

Swiss table with SSE2 SIMD probing and per-slot seqlock concurrency, wrapped by ConcurrentARC which provides ARC eviction policy with hash-keyed O(1) list tracking and deferred promotions via a lock-free MPSC buffer (PromoteBuf). Phase 2a results show a 58% improvement in concurrent Set throughput (per-slot CAS eliminates shared_mutex serialization), safe concurrent reads during writes (resolving Phase 1's unordered_map data race), and a 6–41% single-threaded latency regression (architectural cost of concurrent data structures). A cross-VM baseline comparison shows Furrballs maintains a 2.60x concurrent Set advantage through lock partitioning via per-node CMaps.

Real hardware validation on AWS c6i.metal (Intel Xeon Platinum 8375C, 2 sockets, 128 vCPUs, NUMA distance 10/20) confirms the cross-domain signal on this NUMA instance: 4.5–7.0% p50 cross-domain overhead, thread-local routing provides +60–65% improvement over round-robin, and ablation step D (thread-local routing) is the primary driver of the cross-domain signal (+21.0–38.9% Cross-OH). Get throughput scales +72% from 4→64 threads, validating the lock partitioning thesis for reads. Set throughput degrades -79% at 64 threads due to per-domain SpinLock contention, identifying striped SpinLocks or per-bucket ARC locks as the next optimization target. Cross-platform validation on AWS c6a.metal (AMD EPYC 7R13 Milan, 4 NUMA nodes, 192 vCPUs, NUMA distance 10/12/32) confirms the benefit is vendor-independent: FurrBallTL achieves 1.5x lower GET latency than CacheLib at 64B and 9.5x lower at 128MB working-set (UniformRO 1024B), with 100% vs 50% memory efficiency. CacheLib's per-pool NUMA routing is ineffective on both platforms.

Phase 1 benchmark results on a QEMU NUMA simulation (2 nodes, 4 vCPUs) demonstrated that lock-free reads expose an 11.7% p50 and 13.3% p99 cross-domain overhead—more than double the signal visible under shared_mutex (5.1% p50). Thread-local routing achieves 26–30% p50 and 31–41% p99 improvement over round-robin with lock-free reads, confirming that topology-aware placement provides compound benefits in both memory locality and synchronization reduction. A cross-VM baseline comparison reveals that per-domain sharding provides a 3x concurrent Set and 1.2x concurrent Get throughput advantage through natural lock partitioning, at a mixed single-threaded cost (Set +3%, Get +17%). A five-step ablation study isolates each architectural decision: topology-aware allocation produces no signal in QEMU, per-domain sharding serves as an architectural enabler, thread-local routing creates the first measurable locality signal (+18.3% cross-domain overhead), and a shared-nothing variant using an MPSC slot queue adds +114% cross-domain overhead (~1,500ns queue round-trip) with a break-even at ~21 cache misses per operation on Xeon hardware.

1. Problem Statement

Modern servers expose asymmetric memory topologies — NUMA, CXL-attached memory, HBM tiers, persistent memory — where memory access latency depends on which domain the data resides on. Cross-domain access can be 1.2–10x slower than local access, with the asymmetry increasing as CXL and far-memory technologies deploy. NUMA is the most widespread instance of this asymmetry today (2-socket servers, 1.2–2x latency ratio); CXL-attached memory (3–5x) and

tiered memory architectures (5–10x) represent the emerging and more challenging cases. Existing caching systems either ignore memory topology entirely (Redis, Memcached) or handle it at coarse granularity (Meta's CacheLib uses static per-domain sharding). To our knowledge, no published system combines asymmetric memory topology as an adaptive input to per-page cache placement decisions.

Furrballs targets this gap: allocate cache pages on the memory domain closest to the thread that accesses them, measure the locality benefit, and provide a structured architecture for evolving the policy across topology types.

2. Design

2.1 Architecture Overview

```
L1: ConcurrentARC<KeyMeta>    -- Key-level ARC with CMap backing (Phase 2a, current)
L2: ARC<PageIndex, Page*>     -- Page-level cache, tracks hot pages (Phase 2b)
L3: RocksDB                   -- Persistent storage, written on eviction/shutdown
```

Phase 2a: CMap-based concurrent key store with ARC policy via ConcurrentARC. Pages are never evicted from memory; the ARC lists track key hot/cold state but eviction callbacks are not yet wired. All inserted keys remain resident until shutdown.

2.1.1 Locality Unit

The **page** is the atomic locality unit in Furrballs. All NUMA placement decisions operate at page granularity: a page is allocated on a specific NUMA node via `Numatic::AllocateOnNode()`, and all data within that page shares the same NUMA affinity. Keys are packed into pages via the bump allocator (§2.4), and each key's `KeyMeta` records which page it belongs to (`PageIndex`) and which node the page resides on (`NodeID`).

The separation of locality unit (page) from access unit (key) is deliberate: it allows the eviction policy (Phase 2) to make decisions at page granularity—evicting an entire page reclaims memory and resets the bump allocator—while the application operates at key granularity via `Set`/`Get`. Phase 2b replaced the planned two-level ARC hierarchy (L1 key-level + L2 page-level) with a single unified REMARC policy (§3.29), where per-key REMARC scores drive both key-level and page-level decisions from the same state. The locality signal flows through the unified policy: a page with many cross-domain key accesses accumulates higher remote scores, making it a preferred eviction candidate.

2.1.2 Cross-Node Access Behavior

When a `Get()` accesses data on a remote NUMA node, Furrballs **tracks the cross-node access and defers action to the eviction policy**. Phase 2a's ConcurrentARC (L1) tracks per-key access frequency via ARC list membership (t1/t2/b1/b2) and provides eviction callbacks, but does not yet record per-key cross-node access counts. Phase 2b adds `CrossNodeAccesses` (atomic counter in `KeyMeta`) to `Get()`: when the accessing thread's node differs from the key's `NodeID`, the counter is incremented with `fetch_add(1, relaxed)`. When `CrossNodeAccesses` exceeds a configurable threshold, the key is a candidate for cross-node migration: the key's CMap entry is removed from the source node and re-inserted on the accessing node's CMap, with `memcpy` of the value data to a page on the destination node. This moves the key's data closer to the threads that access it, eliminating repeated cross-node latency.

This design choice (track-and-defer rather than immediate migration or replication) avoids the complexity and consistency overhead of runtime data movement, deferring the decision to the eviction boundary where it is cheapest. Cross-node migration is rare (high threshold) compared to intra-node migration (compaction of cold keys to cold pages, §2.9).

2.1.3 Locality Terminology

Throughout this document, "local" is ambiguous without a frame of reference. We define three terms:

- **Owner-local**: Memory that resides on the same NUMA node as the thread that allocated it (the owner). Under thread-local routing, the owner is the writing thread's node.
- **Requester-local**: Memory that resides on the same NUMA node as the thread currently accessing it (the requester).
- **Remote-to-requester**: Memory that resides on a different NUMA node than the requester's node.

Under thread-local routing, a self-read (requester == owner) is both owner-local and requester-local. A cross-read (requester owner) is owner-local but remote-to-requester.

2.2 Memory Allocation

At initialization (`CreateBall`), one physical allocation is made per NUMA node via `Numatic::AllocateOnNode()`. This block is subdivided into logical pages (each `PageSize` bytes). All logical pages are pre-populated in cache.

```
// Per node:
physicalBlock = Numatic::AllocateOnNode(physicalPageSize, nodeId)
for i in 0..numPages:
```

```
    page[i].Data = (char*)physicalBlock + i * config.PageSize
    page[i].NodeId = nodeId
```

Container allocations within nodes use `std::pmr::memory_resource` backed by `NumaLocalMemoryResource`, ensuring vector backing arrays land on the correct node.

2.3 Key-Based API

```
Error Set(const std::string& key, void* data, size_t size);
Error Get(const std::string& key, void* outBuf, size_t bufSize, size_t& outSize);
```

Key routing supports two strategies, selectable at runtime via `NumaConfig::UseThreadLocalRouting`: round-robin distribution via `AtomicRoundRobin` (default), and thread-local placement via `Numatic::GetCurrentNode()`. Both strategies are topology-aware: round-robin distributes keys evenly across domains, while thread-local places keys on the requesting thread's domain for maximum locality. The trade-off is that `Get()` must search all node shards when the key's location is unknown—a cost that will be eliminated by a key routing table in Phase 4.

2.4 Bump Allocator

Multi-value pages use an append-only bump allocator with 8-byte alignment. Each page tracks `UsedBytes` (atomic, CAS-based concurrent bump) and `DataWastedByPadding` for fragmentation metrics. No per-value deallocation — pages are recycled wholesale (§2.9).

2.4.1 Page Lifecycle

Pages transition through four states:

State	Description
HOT	Receives new `Set()` writes. Active bump allocator. May contain cold keys awaiting compaction.
COLD	Receives migrated values from HOT pages only. Bump allocator advancing from compaction traffic.
EMPTY	No active keys. Available for reuse as HOT or COLD. Bump allocator reset.
FREEZE	Mid-eviction. No writes allowed. Page data being flushed to persistent storage (§2.9).

Transitions: HOT → EMPTY (all keys migrated out by compactor) → COLD (reused as migration destination). COLD → FREEZE (eviction triggered) → EMPTY (after persistent storage flush). Pages recycle indefinitely — no new physical allocation needed after initialization.

`PageTier` is `std::atomic<PageTier>` to prevent `Set()` from writing to a page mid-eviction (the `FREEZE` check adds one branch to the bump path, negligible in the common case).

2.4.2 Page Metadata (Phase 2b)

```
struct Page {
    char* Data;
    uint8_t NodeId;
    std::atomic<size_t> UsedBytes;
    std::atomic<size_t> DataWastedByPadding;

    // Phase 2b:
    std::atomic<bool> HasColdKeys;       // ARC eviction callback sets this
    std::atomic<uint16_t> ActiveKeys;    // keys currently live on this page
    SpinLock CompactLock;                // protects KeyIndex, TempCtrl modifications
    std::vector<HashPair> KeyIndex;      // reverse index: which keys live here
    std::vector<uint8_t> TempCtrl;       // parallel to KeyIndex: 0x00=HOT, 0x01=COLD
    std::atomic<PageTier> Tier;          // HOT, COLD, EMPTY, FREEZE
    void ResetBump();                    // resets UsedBytes and DataWastedByPadding to 0
};
```

`KeyIndex` and `TempCtrl` are parallel arrays: `TempCtrl[i]` corresponds to `KeyIndex[i]`. `TempCtrl` encodes each key's temperature in a byte (inspired by Swiss table ctrl bytes, §2.7), enabling SIMD-scannable cold key detection by the compactor (§2.9). Both arrays are protected by `CompactLock` (per-page SpinLock): the eviction callback and promotion callback acquire it under ConcurrentARC's SpinLock, the compactor acquires it during migration, and cooperative `Set()` migration acquires it. Lock ordering is consistent: `CompactLock` is always taken after ConcurrentARC's SpinLock (never before), preventing deadlock.

2.5 Concurrency

Phase 1 used per-node `shared_mutex` for writes and `SeqLock<KeyMeta>` for lock-free reads under warmup-then-read workloads. The `unordered_map::find()` data race during concurrent reads+writes was a known limitation.

Phase 2a replaces the per-node `unordered_map` + `SeqLock` + `shared_mutex` combination with CMap (§2.7) inside ConcurrentARC (§2.8). The concurrency model is now:

- **Find (Get):** Lock-free CMap read via seqlock protocol. PromoteBuf enqueue for deferred ARC promotion (lock-free `fetch_add`). No SpinLock on the read path. Phase 2b adds `CrossNodeAccesses.fetch_add(1, relaxed)` when the accessing thread's node differs from the key's `NodeID`.
- **Set (insert/update):** UpdateInPlace fast path (CMap seqlock only, no SpinLock) for existing keys that fit. New-key path: SpinLock for ARC list management + CMap for concurrent key→value storage. CMap's per-slot CAS eliminates shared_mutex serialization. Phase 2b adds cooperative migration: every 16th `Set()` on a page with `HasColdKeys == true` migrates one cold key to a COLD page (§2.9).
- **Eviction:** SpinLock + CMap's `FindAndEraseByHash` (hash-keyed, no string needed). Phase 2b's eviction callback sets `page.HasColdKeys = true` and updates `TempCtrl` (§2.9). No I/O on the eviction path — persistent storage writes happen during cold page eviction (whole-page flush), not during ARC eviction.
- **Compaction (Phase 2b):** NodeJob worker(s) scan pages with `HasColdKeys == true`, SIMD-scan `TempCtrl` for cold keys (0x01), migrate values to COLD pages via `CMap.UpdateInPlace`. Background compaction amortized across idle cycles; cooperative migration on the Set path provides additional throughput under load (§2.9).

Statistics use `alignas(64)` on all atomic fields to prevent false sharing. `Numatic::GetCurrentNode()` uses a `thread_local` cache (§3.15) to eliminate syscall overhead per routing decision.

2.6 Platform Abstraction (Numatic)

All NUMA operations are isolated behind stateless free functions:

Category	Functions
Topology	`IsNUMAAvailable`, `GetNodeCount`, `GetDistance`
Threading	`PinCurrentThreadToNode`, `GetCurrentNode`
Allocation	`AllocateOnNode`, `AllocateLocal`, `FreeNUMA`
Huge Pages	`AllocateOnNodeHuge`, `AllocateLocalHuge`, `IsHugePagesAvailable`, `GetHugePageSize`
Migration	`MovePages`, `MigratePages`, `GetPageNode`
PMR	`NumaLocalMemoryResource`, `NumaNodeMemoryResource` (CRTP base)

Linux implementation uses libnuma. Windows has stubs/heuristics for functions without direct equivalents.

2.7 CMap: Concurrent Swiss Table

CMap is a lock-free-read, CAS-based-write concurrent hash map that replaces the Phase 1 `unordered_map<string, unique_ptr<SeqLock<KeyMeta>>>` per-node key store. It eliminates per-key SeqLock overhead, removes the data race on `unordered_map::find()` during concurrent reads, and serves as the backing map inside ConcurrentARC (§2.8). Phase 2a implements and benchmarks the full CMap + ConcurrentARC stack.

2.7.1 Design Constraints

- `alignas(64)` invariant: no false sharing on hot paths
- Lock-free reads: Get() holds zero locks
- No stored keys: 128-bit hash split into H1 (64-bit, group index) + H2 (64-bit, fingerprint). Collision = cache miss, acceptable for a cache. 64-bit fingerprint collision probability: ~2^-64 per pair. Page metadata stores full `HashPair{h1, h2}` (originally stored truncated `uint32_t` h2, fixed after discovering h2 truncation caused false-positive key lookups during staging relocation).
- Fixed capacity, no resize (Phase 2a)
- No iteration (ARC tracks entries externally)
- Topology-blind: memory provided by caller at construction
- Value must be trivially copyable (required for seq-based memcpy reads) and move constructible (required for future resize)

2.7.2 Data Layout

Two separate allocations via `CMapAllocFn` (caller-provided allocator, defaulting to `::operator new` with `std::align_val_t{64}`):

Ctrl array (1 byte/slot, dense):

- `0x80` = EMPTY, `0xFE` = DELETED, `0xFF` = SENTINEL (after last group)
- `0x00--0x7F` = OCCUPIED (lower 7 bits = h2_short, top 7 bits of H2)
- Values `0x81--0xFD` are structurally unreachable: h2_short occupies only the lower 7 bits (0–127), so OCCUPIED ctrl bytes are always `<= 0x7F`. The only special values above `0x80` are `0xFE` (DELETED) and `0xFF` (SENTINEL).
- Modified only on INSERT (CAS EMPTY→OCCUPIED) and DELETE (CAS OCCUPIED→DELETED). Not modified during UPDATE.
- Enables SIMD probing: 16 consecutive bytes per group, single SSE2 load.

Slot array (cacheline-aligned per slot):

- `seq` (1 byte, `atomic<uint8_t>`) + pad (7) + fingerprint (8 bytes, `atomic<uint64_t>`) + Value (N bytes)

- Compile-time layout decision: if `16 + sizeof(Value) <= 64`, inline (one cacheline per slot); if `16 + sizeof(Value) > 64`, split into metadata (64 bytes, cacheline-aligned) + data (`sizeof(Value)`, packed).
- seq uses odd/even: odd = writer active, even = stable.
- `fingerprint` is `std::atomic<uint64_t>` — reads use `.load(relaxed)`, writes use `.store(val, relaxed)`. On x86-64, this compiles to the same instructions as plain `uint64_t` (zero overhead). The seq CAS provides ordering.
- Each slot is `alignas(64)` — no false sharing between adjacent slots.
- Slots are initialized via placement-new (`new (&slots_[i]) Slot{}`) after allocation, ensuring atomic members start in defined state (seq=even/stable, fingerprint=0).

2.7.3 Hashing

Single call to `XXH3_128bits(key)` produces:

- H1 = upper 64 bits → group index: `H1 & (numGroups - 1)`
- H2 = lower 64 bits → stored as fingerprint in slot
- h2_short = top 7 bits of H2 (`H2 >> 57`) → stored in ctrl byte for SIMD pre-filtering

No separate hash function or seed needed. One call, split output.

2.7.4 Probing

Group-based Swiss table probing with SSE2 SIMD (4 intrinsics, 1 header `<emmintrin.h>`):

1. Load 16 ctrl bytes (`_mm_loadu_si128`)
2. Compare against target h2_short (`_mm_cmpeq_epi8`)
3. Extract match bitmask (`_mm_movemask_epi8`)
4. For each matching position: compare full 64-bit fingerprint

Probe termination: if any ctrl byte is EMPTY in the scanned group, the key is guaranteed absent (no false negatives possible). Deleted slots do not terminate probing (key may have been inserted past them).

Template parameter: `Probe<FindDeleted>` — Insert uses `Probe<true>` to record tombstones for reuse; Find/Erase use `Probe<false>` to skip the DELETED SIMD compare entirely (compile-time optimization via `if constexpr`).

Probe pseudocode (deviations from standard Swiss table noted):

```
Probe<FindDeleted>(h1, h2_short, h2):
  deletedSlot = NONE
  for group = h1; ; group++ (mod numGroups):
    mask16 = load_16_ctrl_bytes(group)         // SSE2: _mm_loadu_si128
    match_mask = compare_eq(mask16, h2_short)   // SSE2: _mm_cmpeq_epi8 + _mm_movemask_epi8
```

```
if constexpr FindDeleted:
  del_mask = compare_eq(mask16, 0xFE)       // Extra SIMD compare for tombstone tracking
  for each bit set in del_mask:
    if deletedSlot == NONE: deletedSlot = {group, bit_position}

for each bit i set in match_mask:
  slot = group * 16 + i
  if slots[slot].fingerprint.load(relaxed) == h2:  // Full 64-bit verification
    return {slot, deletedSlot}

empty_mask = compare_eq(mask16, 0x80)
if empty_mask != 0:                            // EMPTY terminates probe
  return {NONE, deletedSlot}
```

Deviation: standard Swiss table (Abseil) stores only h2_short in ctrl and compares inline. CMap stores the full 64-bit H2 as `atomic<uint64_t>` in each slot, adding a second verification level. The ctrl byte pre-filter eliminates 127/128 candidates via SIMD; the remaining ~1/128 matches are verified by the full fingerprint comparison.

2.7.5 Concurrency Model

Two CAS points, one per architectural concern:

CAS on ctrl byte — structural state transitions:

- INSERT: CAS ctrl DELETED → OCCUPIED (h2_short), or EMPTY → OCCUPIED (h2_short)
- DELETE: CAS ctrl OCCUPIED → DELETED

Ctrl bytes are plain `uint8_t`, not `std::atomic<uint8_t>`. CAS uses `__atomic_compare_exchange_n` (GCC/Clang builtin) on plain bytes — the same approach as Google/Abseil's `raw_hash_set`. This is necessary because SIMD loads (`_mm_loadu_si128`) cannot operate on `std::atomic` types; using `std::atomic<uint8_t>` would require dropping SIMD probing or invoking undefined behavior. The `__atomic_*` builtins provide the required atomicity guarantees without preventing SIMD access.

CAS on seq counter — writer mutual exclusion + reader versioning:

- Any mutation: CAS seq even→odd (acquire), write data, seq odd→even (release)
- Readers: load seq (acquire), read data, re-check seq. If changed → retry.

Every mutation path acquires seq first:

- INSERT: CAS seq → write fingerprint → CAS ctrl → write value → seq even. Fingerprint is written BEFORE ctrl CAS; the ctrl CAS (ACQ_REL) acts as the publication barrier for fingerprint on x86-64 TSO.
- UPDATE: CAS seq → write fingerprint + value → seq even
- DELETE: CAS seq → CAS ctrl → zero data → seq even

Readers (Find): probe ctrl (lock-free, benign stale reads accepted); on candidate: check seq (acquire), memcpy value, re-check seq; return copy of value (`std::optional<Value>` via `alignas(Value) std::byte buf[sizeof(Value)]` + `std::bit_cast<Value>(buf)`).

`Set()` returns `CMapSetResult{Error err, bool inserted}` so callers know INSERT vs UPDATE from a single probe, eliminating the need for a separate `Find()` before `Set()`.

`UpdateInPlace(key, fn)` acquires the slot's seqlock, calls `fn(value)`, releases seqlock. The callback can modify the Value AND perform side effects (e.g., memcpy to page data) under the seqlock.

`FindAndErase(key)` performs a single probe that reads the value under seqlock and marks the slot deleted, returning `FindAndEraseResult {Error, optional<Value>}` distinguishing KEY_NOT_FOUND from ABANDONED_SET.

`FindAndEraseByHash(HashPair)` accepts a pre-computed hash instead of a string key, used by ConcurrentARC for hash-keyed eviction without needing original key strings.

Ctrl array is mostly read-only after insert — only INSERT and DELETE modify ctrl bytes. UPDATE leaves ctrl untouched, preserving SIMD probing efficiency without false sharing on the read path.

2.7.6 Tombstone Reuse

Deleted slots (`0xFE`) are reused on subsequent inserts. The probe records the first deleted slot encountered; Insert prefers it over empty slots, preventing tombstone accumulation. This is standard Swiss table behavior.

2.7.7 Allocator Interface

CMap uses `CMapAllocFn = void*(*)(size_t)` and `CMapFreeFn = void(*)(void*, size_t)` function pointers, defaulting to `::operator new` / `::operator delete` with `std::align_val_t{64}`. For topology-aware allocation, callers pass `Numatic::AllocateLocal` and `Numatic::FreeNUMA`. This keeps CMap topology-blind — memory placement is the caller's responsibility.

2.7.8 Fail-Fast Contention Model

If CAS on seq fails (another writer is active), the operation returns `ABANDONED_SET` immediately — no spinning. If CAS on ctrl fails after seq was acquired, seq is released (store even) before returning the error. This fail-fast design avoids writer starvation and leaves contention resolution to the caller (ConcurrentARC handles retries at a higher level).

2.7.9 Operation Pseudocode

Find (lock-free read via seqlock):

```
Find(key):
  {h1, h2} = XXH3_128bits(key)
  h2_short = h2 >> 57

  {slot, _} = Probe<false>(h1, h2_short, h2)
  if slot == NONE: return nullopt

  // Seqlock read: retry if writer is active or intervened
  retry:
    seq1 = slots[slot].seq.load(acquire)
    if seq1 is odd: goto retry                    // Writer active
    fingerprint = slots[slot].fingerprint.load(relaxed)
    memcpy(value_buf, &slots[slot].value, sizeof(Value))
    atomic_thread_fence(acquire)
    seq2 = slots[slot].seq.load(relaxed)
    if seq1 != seq2: goto retry                   // Writer intervened
  return bit_cast<Value>(value_buf)
```

Set (single-probe INSERT or UPDATE with dual CAS):

```
Set(key, value) -> CMapSetResult{err, inserted}:
  {h1, h2} = XXH3_128bits(key)
  h2_short = h2 >> 57

  {slot, deletedSlot} = Probe<true>(h1, h2_short, h2)

  if slot != NONE:
    // UPDATE: existing key, ctrl unchanged
    if !CAS(slots[slot].seq, even -> odd): return {ABANDONED_SET, false}
    slots[slot].fingerprint.store(h2, relaxed)
    memcpy(&slots[slot].value, &value, sizeof(Value))
    slots[slot].seq.store(seq + 1, release)       // odd -> even
```

```
    return {OK, false}                                  // inserted=false

  // INSERT: new key, must claim a slot
  target = deletedSlot if deletedSlot != NONE else first_empty_from_probe
  if target == NONE: return {CACHE_FULL, false}

  // Step 1: Acquire seq (writer exclusion)
  if !CAS(slots[target].seq, even -> odd): return {ABANDONED_SET, false}

  // Step 2: Write fingerprint BEFORE ctrl CAS (publication barrier)
  slots[target].fingerprint.store(h2, relaxed)

  // Step 3: CAS ctrl (structural claim) - try tombstone first, then empty
  if !CAS(ctrl[target], DELETED -> h2_short) &&
     !CAS(ctrl[target], EMPTY   -> h2_short):
    slots[target].seq.store(seq + 1, release)     // Release seq on ctrl failure
    return {ABANDONED_SET, false}

  // Step 4: Write value, release seq
  memcpy(&slots[target].value, &value, sizeof(Value))
  slots[target].seq.store(seq + 1, release)       // odd -> even
  return {OK, true}                                  // inserted=true
```

Deviation: standard Swiss tables use a single atomic claim on the ctrl byte. CMap adds seq CAS for writer mutual exclusion + reader versioning, and writes fingerprint before ctrl CAS to ensure the fingerprint is visible when the ctrl transition publishes the slot.

UpdateInPlace (seqlock-protected callback):

```
UpdateInPlace(key, fn) -> Error:
  {h1, h2} = XXH3_128bits(key)
  {slot, _} = Probe<false>(h1, h2 >> 57, h2)
  if slot == NONE: return KEY_NOT_FOUND

  if !CAS(slots[slot].seq, even -> odd): return ABANDONED_SET
  fn(slots[slot].value)                           // Caller modifies value in-place
  slots[slot].seq.store(seq + 1, release)         // odd -> even
  return OK
```

The callback `fn` receives a mutable reference to the slot's value. It can modify the Value AND perform side effects (e.g., `memcpy` into page data at a specific offset) — all under the seqlock's protection. This eliminates the torn-read bug where page data was written outside any atomic window.

FindAndErase (single-probe atomic read + delete):

```
FindAndErase(key) -> {Error, optional<Value>}:
  {h1, h2} = XXH3_128bits(key)
  {slot, _} = Probe<false>(h1, h2 >> 57, h2)
  if slot == NONE: return {KEY_NOT_FOUND, nullopt}

  if !CAS(slots[slot].seq, even -> odd): return {ABANDONED_SET, nullopt}

  value_copy = memcpy_read(slots[slot].value)     // Read under seqlock

  if !CAS(ctrl[slot], OCCUPIED -> DELETED):
    slots[slot].seq.store(seq + 1, release)       // Release seq on ctrl failure
    return {ABANDONED_SET, nullopt}

  memset(&slots[slot].value, 0, sizeof(Value))    // Zero data
  slots[slot].fingerprint.store(0, relaxed)        // Zero fingerprint
  slots[slot].seq.store(seq + 1, release)         // odd -> even
  return {OK, value_copy}
```

FindAndEraseByHash is identical but accepts `{h1, h2}` directly, skipping the xxHash call. Used by ConcurrentARC for hash-keyed eviction from ArcList entries (which store `HashPair`, not original key strings).

2.8 ConcurrentARC

ConcurrentARC wraps CMap + four ARC lists + SpinLock + PromoteBuf into a per-node ARC cache policy. It replaces the Phase 1 flat KeyStore, providing key-level hot/cold tracking with eviction support. **Note:** ConcurrentARC is the Phase 2a policy. Phase 2b replaces it with REMARC (§3.29), which collapses the ARC hierarchy and TempCtrl-based compaction into a single unified mechanism. The ConcurrentARC infrastructure (ArcList, PromoteBuf, SpinLock) remains as the Phase 2a implementation baseline.

2.8.1 ArcList: Hash-Keyed Doubly-Linked List

ARC requires O(1) membership check, move-to-front, and pop-back on four lists (t1, t2, b1, b2). A naive `std::list<string>` + `std::find` is O(n). ArcList uses `std::list<HashPair>` (16 bytes per entry) + `std::unordered_map<uint64_t, iterator>` (H2 as map key) for O(1) `contains`, `push_front`, `pop_back`, `erase`, and `splice_front`. No string storage anywhere in ARC tracking.

2.8.2 PromoteBuf: Deferred MPSC Promotions

Find() must promote the accessed key in ARC lists, but taking a SpinLock on every Get() was the dominant Phase 2a regression (§5.13). PromoteBuf is a fixed-size circular MPSC buffer (256 slots × 16 bytes = 4KB) that defers ARC promotions:

- Find() enqueues a `HashPair` (16 bytes, no string copy) via `fetch_add` on a write head
- Every 64th enqueue signals the caller to `try_lock` + drain (cooperative drain)
- Set() always drains at start under its existing SpinLock
- The system is self-balancing: more threads = more frequent drain attempts = faster drain. Drain rate equals fill rate regardless of thread count.

Overflow behavior: If the circular buffer wraps before a slot is drained, `enqueue()` checks the slot's `ready` flag. If the slot is still occupied, the enqueue returns false and the promotion is lost. Lost promotions are acceptable — a missed promotion means the key stays in its current ARC list position rather than being promoted, which is functionally correct (the key remains cache-resident, just with a slightly stale recency ranking). The next access will attempt a new promotion.

Stale promotions: During drain, a promotion may target a key already evicted by another thread's concurrent `Set()`. The drain logic checks `t1.contains(h2)` and `t2.contains(h2)` before promoting; if neither list contains the key, the promotion is silently discarded. This is the correct behavior — the key no longer exists in the cache, and promoting a ghost entry would corrupt ARC state.

2.8.3 SpinLock

TAS lock with `_mm_pause()` backoff. Provides `lock()`, `try_lock()`, `unlock()`. Used for ARC list mutations (t1/t2/b1/b2 operations) and PromoteBuf drain. Not held during CMap operations — CMap has its own concurrency via seqlock + CAS.

2.8.4 Operations

Find(key): `store_.Find()` (lock-free CMap read) + `promoteBuf_.enqueue(hash)` (lock-free). Cooperative `try_lock` drain every 64th call. Returns `std::optional<Value>`. Phase 2b: `CrossNodeAccesses.fetch_add(1, relaxed)` if accessing from a different node than the key's `NodeID`.

Set(key, val): SpinLock + drain buffer + single `CMap.Set()` probe + hash-keyed ARC list management (standard ARC algorithm using H2 for list tracking). Returns `CMapSetResult`. Phase 2b: cooperative migration — every 16th `Set()` on a page with `HasColdKeys == true` migrates one cold key to a COLD page (§2.9).

UpdateInPlace(key, fn): Delegates to `CMap.UpdateInPlace()` — no SpinLock needed (CMap handles its own concurrency).

Eviction: `replaceLocked(h2)` and `evictLocked()` return bool — false on CAS contention, callers propagate as ABANDONED_SET. List mutation happens AFTER confirmed `FindAndEraseByHash`. Phase 2a EvictionCallback receives `const Value&` (KeyMeta contains all metadata: PageIndex, DataSize, DataOffset, NodeID). Phase 2b changes the signature to `void(uint64_t h2, const Value&)` to carry the hash needed for `TempCtrl` index lookup.

Phase 2b eviction callback: When ARC evicts a key (t1/t2 → b1/b2), the callback acquires `page.CompactLock`, looks up the HashPair in `page.KeyIndex` to find index `i`, sets `page.TempCtrl[i] = 0x01` (COLD), releases `CompactLock`, then sets `page.HasColdKeys = true`. This flags the page for the compactor (§2.9). The callback runs under ConcurrentARC's SpinLock, so `CompactLock` is always taken after the ARC SpinLock (consistent lock ordering). No I/O, no allocation, no blocking operation on the callback path.

Phase 2b promotion callback: When PromoteBuf drain promotes a key (b1/b2 → t1/t2), the callback acquires `page.CompactLock`, looks up the HashPair in `page.KeyIndex` to find index `i`, sets `page.TempCtrl[i] = 0x00` (HOT), releases `CompactLock`. This prevents the compactor from unnecessarily migrating a re-promoted key. The promotion drain already holds the ARC SpinLock, so `CompactLock` is taken after the ARC SpinLock (same ordering as eviction).

2.8.5 Integration with Furrballs

`PerNodeDetails` was restructured: `rwMutex`, `KeyShard`, and `KeyMetaStore` removed; replaced by `ConcurrentARC<KeyMeta> KeyStore` and `std::atomic<size_t> CurrentPage` (lock-free page advancement). `Set()` tries `UpdateInPlace` first (fast path — in-place write under CMap's seqlock, no allocation, no SpinLock); falls back to atomic page allocation + `ConcurrentARC.Set()`. `Get()` calls `ConcurrentARC.Find(key)` per node. CreateBall passes `Numatic::AllocateLocal/FreeNUMA` for NUMA-local CMap memory. Phase 2b: `KeyMeta` gains `Dirty` flag (set on `Set`, cleared on persistent storage flush) and `CrossNodeAccesses` atomic counter (incremented on cross-node `Get`).

2.9 Compactor and Migration-Based Eviction

Phase 2b introduces a migration-based eviction model that replaces per-key eviction from bump-allocated pages with value migration between pages, followed by wholesale page eviction. This solves the fundamental constraint of the append-only bump allocator: per-key eviction leaves holes that cannot be reclaimed without compaction. **Note:** The compactor, evictor, and migration scanner described here are unified under REMARC (§3.29) in Phase 2b — a single scanner operating on REMARC Evict/Migrate scores replaces the separate ARC-driven cold-flagging and compaction mechanisms.

2.9.1 Migration Model

Values flow from HOT pages to COLD pages via migration, and COLD pages are evicted wholesale to persistent storage. Pages recycle between states (§2.4.1).

```
[HOT PAGE]  ARC marks key cold → [HasColdKeys flag set]

                                    [compactor scans TempCtrl]

                                    [memcpy value to COLD page]
                                    [CMap.UpdateInPlace: new PageIndex/Offset]

                                  [COLD PAGE]  page full or memory pressure → [FREEZE → persi


                              [ghost hit: reload from storage → HOT PAGE]
```

Why migration over per-key eviction: Per-key eviction from a bump-allocated page leaves holes in the page. Reclaiming those holes requires compaction (copy remaining values, update all KeyMetas) — expensive and constant under eviction pressure. Migration avoids holes entirely: values are copied to a receiving page, and the source page's wasted space is reclaimed only when the entire page is emptied and recycled. The worst case is temporary memory overhead proportional to pages "in transit," which is bounded and self-correcting.

2.9.2 Compactor Trigger: Hybrid ARC + Background + Cooperative

The compactor uses a hybrid trigger model:

1. **ARC signals (free):** When ARC evicts a key (t1/t2 → b1/b2), the eviction callback sets `page.HasColdKeys = true` and writes `TempCtrl[i] = 0x01`. Zero-cost on the eviction path (atomic store under existing SpinLock).
2. **Background compaction:** NodeJob worker(s) scan pages with `HasColdKeys == true` between dispatches. The scan interval is shorter under memory pressure and longer when memory is plentiful. Self-balancing: idle workers = more frequent compaction.
3. **Cooperative migration (Set path):** Every 16th `Set()` on a page with `HasColdKeys == true` migrates one cold key as a side effect before bumping. Under high write throughput, N threads provide free compaction labor. Throttled to avoid significant latency impact on the write path.

2.9.3 Compactor Algorithm

```
compactor_run(node):
  for each page where page.HasColdKeys == true && page.Tier != FREEZE:
    page.CompactLock.lock()
    // SIMD scan TempCtrl for COLD keys (0x01)
    for each batch of 16 TempCtrl bytes:
      remaining = min(16, TempCtrl.size() - batch)
```

```
      cold_mask = _mm_cmpeq_epi8(TempCtrl[batch], 0x01)
      cold_mask &= (1 << remaining) - 1  // mask final batch to avoid overread
      for each bit i set in cold_mask:
        hashPair = page.KeyIndex[batch + i]
        keyMeta = CMap.Find(hashPair)
        if keyMeta has_value:
          coldPage = find_cold_page_with_space(node)
          if coldPage found:
            newOffset = coldPage.TryBump(keyMeta.DataSize)
            if newOffset valid:
              // CRITICAL: memcpy inside UpdateInPlace to prevent data loss.
              // A concurrent Set() between an external memcpy and UpdateInPlace
              // would overwrite source data. Under seqlock, both the read and
              // the metadata update are atomic.
              CMap.UpdateInPlace(hashPair, [&](KeyMeta& km) {
                memcpy(coldPage.Data + newOffset,
                       page.Data + km.DataOffset, km.DataSize)
                km.PageIndex = coldPage_index
                km.DataOffset = newOffset
              })
              page.ActiveKeys--
              page.KeyIndex.remove(hashPair)
              coldPage.ActiveKeys++
              coldPage.CompactLock.lock()
              coldPage.KeyIndex.push_back(hashPair)
              coldPage.TempCtrl.push_back(0x00)  // newly migrated = HOT initially
              coldPage.CompactLock.unlock()
    page.CompactLock.unlock()

    page.HasColdKeys.store(false, relaxed)

    if page.ActiveKeys == 0:
      page.Tier.store(EMPTY, relaxed)
      page.ResetBump()
      page.CompactLock.lock()
      page.KeyIndex.clear()
      page.TempCtrl.clear()
      page.CompactLock.unlock()
```

Finding a COLD page with space: Check existing COLD pages for one where `TryBump` succeeds. If none, convert an EMPTY page to COLD. If no EMPTY pages available, migration stalls — the system is self-correcting (eviction of full COLD pages frees EMPTY pages for reuse).

Re-promoted keys: If a key was evicted from ARC (TempCtrl = COLD) then re-promoted via ghost hit (TempCtrl = HOT), the compactor skips it on the next scan. The TempCtrl byte is the single source of truth for "should this key be migrated?" — no SpinLock or ARC membership query needed.

Unnecessary migrations: In the window between ARC eviction and compactor scan, a key might be re-promoted back to t1/t2. If the promotion callback hasn't cleared TempCtrl yet, the compactor migrates it unnecessarily. This is harmless: the key lives on a COLD page but is hot in ARC, and reads from COLD pages have identical latency to HOT pages (same memory, same NUMA node).

2.9.4 Cross-Node Migration

When `KeyMeta.CrossNodeAccesses` exceeds a configurable threshold (e.g., 100–1000 accesses), the key is a candidate for cross-node migration:

```
cross_node_migrate(hashPair, destNode):
  // 1. Copy value data to destination page
  newOffset = destPage.TryBump(keyMeta.DataSize)
  if newOffset invalid: return CACHE_FULL
  memcpy(destPage.Data + newOffset, sourcePage.Data + keyMeta.DataOffset, keyMeta.DataSize)

  // 2. Insert into destination node's CMap FIRST (key now on both nodes)
  result = destCMap.Insert(hashPair, KeyMeta{destPage_index, newOffset, ...})
  if result failed: return error  // dead space on dest page, acceptable bump waste

  // 3. Erase from source node's CMap (retry until success)
  retry:
    result = sourceCMap.FindAndEraseByHash(hashPair)
    if result failed (CAS contention): goto retry

  keyMeta.CrossNodeAccesses = 0
  // Update page bookkeeping on both nodes (under CompactLock)
```

Insert-before-erase ordering: The destination insert happens before the source erase. During the brief window between insert and erase, the key exists on both nodes — concurrent `Get()` calls find it on whichever node they check first, always returning a correct result. No transient miss window, no wrong answers, no forwarding mechanism needed. CMap's per-slot CAS protects both operations independently (insert and erase target different slots on different nodes), so contention probability and surface area are minimal.

Failure handling: If the dest insert fails (CAS contention), the value data occupies dead space on the dest page (reclaimed on page eviction). If the source erase fails (CAS contention), the retry

loop eventually succeeds. During retries, the key is on both nodes — dual ownership is invisible to readers and correct for Gets. A concurrent `Set()` targeting this key during the dual-ownership window would hit the source (pre-erase) or the dest (post-insert) — both correct.

The CMap entry fully relocates to the destination node, restoring the sole-owner invariant after the erase completes. Cross-domain migration is rare (high threshold) and represents the thesis's topology-aware placement contribution: data moves to where the threads that access it are.

2.9.5 Whole-Page Persistent Storage

When a COLD page fills up or memory pressure requires reclamation, the page is evicted to persistent storage:

PageBlob format:

```
struct PageBlob {
    uint8_t NodeId;
    uint16_t ActiveKeys;
    uint16_t KeyCount;
    // Per-key metadata array:
    struct { uint64_t H1; uint64_t H2; uint32_t DataOffset; uint32_t DataSize; bool Dirty; } K
    // Raw page data:
    uint8_t ValueData[UsedBytes];
};
```

Eviction flow: 1. `page.Tier.store(FREEZE, release)` — block new writes 2. Serialize PageBlob from page data + KeyIndex + TempCtrl 3. `RocksDB.Put(PageId, serialized_blob)` — single write per eviction 4. Update `evictedKeyIndex_[h2] = PageId` for each key on the page (in-memory H2 → PageId mapping for ghost hit recovery) 5. `CMap.FindAndEraseByHash(h2)` for each key on the page 6. `page.Tier.store(EMPTY, release)` + `page.ResetBump()` + clear bookkeeping

Why whole-page, not per-key: Single RocksDB write per eviction vs N writes. The page's temporal locality (keys that went cold around the same time) makes whole-page reload on ghost hit speculative but effective (§2.9.6). RocksDB's LSM tree overhead is amortized over larger writes.

2.9.6 Ghost Hit Recovery

Two recovery scenarios:

Scenario A: Key in CMap, in ARC ghost list (b1/b2). - `CMap.Find()` returns the value (data still in memory, on a COLD page) - ARC promotes back to t1/t2 via PromoteBuf drain - `TempCtrl` byte set to HOT (0x00) - No persistent storage I/O

Scenario B: Key NOT in CMap (COLD page was evicted). - `CMap.Find()` returns nullopt on all nodes - Look up `evictedKeyIndex_[h2]` → PageId - `RocksDB.Get(PageId)` → deserialize

PageBlob - Reload entire page into memory: - Allocate an EMPTY page as HOT - `memcpy` the ValueData - Re-insert all keys into CMap with their original KeyMeta - Rebuild page bookkeeping (KeyIndex, TempCtrl, ActiveKeys) - Remove all entries from `evictedKeyIndex_` for this page - Return the requested key's value

Speculative whole-page reload: When one key on an evicted page is accessed, the entire page is reloaded. Since the compactor migrates keys in temporal order (ARC evicts keys by recency), COLD pages contain keys that "went cold around the same time." Accessing one often means others will follow. This is the same principle as OS page fault handling.

2.9.7 Evicted Page Lifecycle Management

Evicted pages accumulate in RocksDB. For the thesis, cleanup strategies:

- **TTL per page:** Each evicted page gets a timestamp. After N seconds, RocksDB compaction deletes it. Access after TTL = true `KEY_NOT_FOUND`.
- **On shutdown:** Full RocksDB cleanup/compaction. Speed is irrelevant at shutdown.
- **Separate process:** Persistent storage maintenance is orthogonal to the thesis contribution (topology-aware in-memory caching). Could be a standalone tool that compacts and prunes RocksDB independently of Furrballs.

The `evictedKeyIndex_` (H2 → PageId) is in-memory only. If the process restarts, the index is lost — but this is a cache, not a database. Cold restart warms up from scratch, which is the expected behavior for a caching layer.

2.10 Staging Pages and Page Drain

Phase 2c introduces staging pages to eliminate the SET tail-latency bottleneck caused by `TryAllocFromFree` scanning all pages after eviction. For policies with `HasStoreEviction = true` (ARC, LRU), the last page in each node's page pool is designated as a staging page.

2.10.1 The SET Bottleneck

Without staging, the SET path under capacity pressure follows this sequence:

1. `TryBump()` on CurrentPage fails (page full)
2. Advance CurrentPage to next page; if all pages exhausted, enter fallback
3. Fallback: `ForceEvictOne()` evicts one key via ARC
4. `TryAllocFromFree()` scans all pages for the freed slot — O(P) where P = pages per node
5. Repeat if scan finds nothing

Step 4 is the bottleneck. After evicting one key in O(1), the SET thread scans all pages looking for the freed space. With 8192 pages per node, this scan dominates p99_set latency (68K ns observed on c6i.metal).

2.10.2 Staging Page Design

The staging page provides an O(1) overflow target for SET, decoupling the write path from space management:

```
SET path (HasStoreEviction policies):
  1. TryBump on CurrentPage        → success: done (O(1))
  2. Advance CurrentPage (skip Staging pages in rotation)
  3. If all pages exhausted:
     a. TryBump on Staging page    → success: done (O(1))
     b. ForceEvictOne + TryAllocFromFree scan (fallback, rare)
```

Staging page properties:

- Same `PageTier::Staging = 5` enum value, accepted by `TryBump` (Hot or Staging)
- Allocated from the same physical page pool as regular pages (no separate allocation)
- Never selected as CurrentPage during normal rotation
- Bump allocator only — no `TryAllocFromFree` scanning on staging

SET correctness: A key written to the staging page is fully valid. Its `KeyMeta` records the staging page index. GET reads it via the same path as any other page. The staging page's bump allocator advances independently of the regular page rotation.

2.10.3 Background Relocation

The maintenance thread (NodeJob, 2ms interval) relocates keys from the staging page back to regular pages:

```
BackgroundEvict Phase 1 (staging relocation):
  for i in 0..64:
    if stagingPage.ActiveKeys == 0: break
    hash = stagingPage.KeyH1[0], KeyH2[0]   // first key in staging
    if relocateKey(hash, skipPage=SIZE_MAX):  // move to any regular page
      stagingPage.RemoveKeyByHash(hash)
    else:
      KeyStore.ForceEvictOne()               // make room

  if stagingPage.ActiveKeys == 0 && stagingPage.UsedBytes > 0:
    stagingPage.Recycle()                     // reset bump
    stagingPage.Tier = Staging                // keep staging designation
```

`relocateKey` operates under CMap's seqlock: it copies value data to the destination page first, then updates `KeyMeta` atomically. Between copy and update, GET reads the old location (still valid). After update, GET reads the new location (data already written). No torn reads, no locking on the read path.

2.10.4 Page Drain Compaction

Phase 3 of `BackgroundEvict` proactively consolidates sparse pages:

```
BackgroundEvict Phase 3 (page drain):
  threshold: ActiveKeys <= 4 && DeadBytes > PageSize/4
  for each Hot page (excluding staging):
    if sparse enough: drainTarget = page with most dead bytes

  while drainTarget.ActiveKeys > 0:
    hash = drainTarget first key
    if relocateKey(hash, skipPage=drainTarget):
      drainTarget.RemoveKeyByHash(hash)
    else:
      ForceEvictOne(); break

  if drainTarget empty:
    drainTarget.Recycle()
    drainTarget.Tier = Hot
    CurrentPage = drainTarget    // fresh bump target for SET
```

This creates a virtuous cycle: eviction empties sparse pages → drain consolidates remaining keys → recycled page becomes a fresh bump target → SET uses fast bump (O(1)) instead of staging or scanning.

2.10.5 Destructor Synchronization

Each FurrBall maintains `Destroying` (atomic bool) and `ActiveMaintenanceRefs` (atomic int). The destructor sets `Destroying = true` before removing from the global `OpenBalls` list, then spins until `ActiveMaintenanceRefs == 0`. The maintenance lambda uses a double-check pattern: check `Destroying`, increment refs, re-check `Destroying`, process, decrement refs. This replaces the heuristic 20ms sleep and prevents use-after-free when the maintenance thread holds a snapshot of OpenBalls during ball destruction.

3. Design Decisions

3.1 Per-Node Physical Block with Subdivision

Decision: One `Numatic::AllocateOnNode()` call per NUMA node at init, manually partitioned into logical pages.

Rationale: Avoids syscall overhead per page. Single allocation + pointer arithmetic is faster than N `numa_alloc_onnode` calls. AMP (Phase 3) will handle expansion to multiple physical blocks.

3.2 PMR-Backed Per-Node Containers

Decision: `PerNodeDetails::NodePages` uses `std::pmr::vector<Page>` backed by `NumaLocalMemoryResource`.

Rationale: Standard `std::vector` allocates on the OS's preferred node. PMR with a NUMA-local resource ensures the backing array memory lands on the correct node, since `PerNodeDetails` is constructed on a pinned worker thread.

3.3 Key Routing Strategy

Decision: Keys are routed via either `AtomicRoundRobin` (default) or `Numatic::GetCurrentNode()` (thread-local), selected at runtime by a `NumaConfig::UseThreadLocalRouting` flag. The dispatch uses a ternary operator: `ThreadLocalRoute ? Numatic::GetCurrentNode() : rr.Get()`.

Rationale: Round-robin provides uniform distribution. Thread-local routing places keys on the requesting thread's node, maximizing read-after-write locality. Both strategies are topology-aware—placement decisions consider topology. The cost is that `Get()` must iterate all node shards when the key's location is not known, which adds overhead proportional to the node count. Hash-based routing (`hash(key) % nodeCount`) was evaluated as an alternative that eliminates shard iteration, but it is topology-blind (placement ignores topology) and was rejected for the thesis because it weakens the topology-aware placement claim. A future key routing table (Phase 4) will provide O(1) lookup while preserving topology-aware placement.

3.4 All-or-Nothing NUMA Init

Decision: If any node fails to allocate during `CreateBall()`, all successful allocations are freed and `nullptr` is returned.

Rationale: Partial topology (e.g., one domain fails, fallback to single-domain) eliminates the asymmetry variable from benchmarks. One domain = uniform memory access = no cross-domain effect to measure.

3.5 WaitGroup for Parallel Init

Decision: NodeJob workers allocate their node's physical block in parallel, synchronized by StreamLine's `WaitGroup`.

Rationale: Init is blocking by design (the FurrBall must be fully constructed or not at all). WaitGroup provides zero-overhead thread synchronization without busy-waiting.

3.6 Error Codes, Not Exceptions

Decision: Public API returns `Error` enum. Exceptions only for unrecoverable programmer errors (WaitGroup misuse).

Rationale: Consistent with systems programming conventions. The `Error` enum is categorized (general, initialization, memory, cache, storage, compression, threading, I/O) for easy filtering.

3.7 `alignas(64)` on Statistics Atomics (Revised)

Decision: Each atomic in `Statistics` sits on its own 64-byte cache line. **Revised (v1.33.0):** Hot-path statistics (`HitCount`, `BytesRead`, `LocalHitCount`, `BytesWritten`) moved to per-node atomics in `PerNodeDetails` (NUMA-allocated on each node). Global `Statistics` preserved for cold-path counters and aggregation.

Rationale (original): Prevents false sharing between concurrent `HitCount`, `MissCount`, `BytesWritten`, and `BytesRead` updates in multi-threaded benchmarks.

Rationale (revision): The original design caused a 6.6x GET degradation at 4T on c6a.metal (4 NUMA nodes, 2 sockets). With `alignas(64)`, each of the 14 counters occupies its own cache line on the FurrBall heap object (node 0). At 4T spanning two sockets, every `fetch_add` requires cross-socket exclusive cache-line transfer (~120ns per line on Infinity Fabric, 3 lines per GET = ~360ns overhead). The per-node sharding eliminates all cross-socket stat contention: 4T p50 GET drops from 587ns to 71ns, matching 1T. **Lesson: `alignas(64)` prevents intra-socket false sharing but maximizes inter-socket contention. NUMA-aware systems must shard performance counters by NUMA domain, not merely isolate them by cache line.**

3.8 Hash-as-Key (No Stored Keys)

Decision: CMap stores a 64-bit fingerprint (H2) instead of the original key string. Lookup is by hash only.

Rationale: Variable-size strings prevent fixed-size slots. A 64-bit fingerprint has collision probability ~2^-64 per pair, effectively zero for cache-scale workloads (~262K entries). Collision = cache miss, functionally indistinguishable from an eviction. Eliminates string storage, string comparison, and per-entry heap allocation.

3.9 Separate Ctrl Array for SIMD Probing

Decision: Control bytes are in a separate dense array (1 byte/slot), not interleaved with slot data.

Rationale: SIMD probing loads 16 consecutive ctrl bytes in one instruction (16 bytes). If ctrl bytes were interleaved in 64-byte slots, loading 16 ctrl bytes would span 768 bytes. The separate

array keeps SIMD loads cache-friendly. The ctrl array is rarely modified (only on insert/delete, not update), so false sharing on the dense ctrl array is minimal.

3.10 Compile-Time Slot Layout

Decision: Slot layout adapts to sizeof(Value) at compile time. Values <= 48 bytes share a cacheline with seq+fingerprint; larger values get a separate aligned region.

Rationale: Forces each slot's seq counter onto its own cacheline (`alignas(64)` invariant), but avoids wasting an entire cacheline for small values that fit inline. The threshold (48 bytes) is derived: seq(1) + pad(7) + fingerprint(8) = 16 bytes overhead, leaving 48 bytes in a 64-byte line.

3.11 CAS on Seq for Writer Serialization

Decision: Writers acquire exclusive access via CAS on the per-slot seq counter (even→odd), not via mutex.

Rationale: CAS seq serializes writers without kernel mutex overhead. The same seq counter serves readers (version check) and writers (mutual exclusion), providing a single mechanism for both R/W and W/W concurrency. The odd/even encoding lets readers detect mid-write states without any lock.

3.12 `inline` Not `static` on Namespace-Scope Functions in Headers

Decision: `HashKey()` uses `inline` linkage at namespace scope, not `static`.

Rationale: `static` in a header creates a separate copy per translation unit (ODR-safe but wasteful). `inline` provides a single definition across TUs, matching the standard library pattern for header-only utilities.

3.13 Hash-Keyed ARC Tracking

Decision: ArcList stores `HashPair` (16 bytes: H1 + H2) in `std::list`, indexed by `unordered_map<uint64_t, iterator>` using H2 as key. No string storage in ARC lists.

Rationale: ARC tracks entry ordering for eviction decisions — it needs O(1) contains/erase/splice_front, not the original key string. Storing `HashPair` instead of `std::string` eliminates all string copies in the ARC path. The `unordered_map` index provides O(1) lookup; the `std::list` provides O(1) reordering. `FindAndEraseByHash(HashPair)` enables eviction without the original key string.

3.14 PromoteBuf Deferred Promotion with Cooperative Drain

Decision: Find() enqueues `HashPair` (16 bytes) into a fixed-size circular MPSC buffer. Every 64th enqueue signals the caller to `try_lock` + drain. Set() always drains at start under its existing SpinLock.

Rationale: Taking SpinLock on every Get() was the dominant Phase 2a regression (concurrent Get dropped to 0.56x vs baseline). Deferred promotions make Find() fully lock-free. The cooperative drain is self-balancing: more threads = more frequent drain attempts = faster drain, regardless of thread count. The 64-slot interval is tuned for low contention; per-thread sharding is the documented upgrade path for 100+ threads per node.

3.15 `thread_local` Cache for `GetCurrentNode()`

Decision: `Numatic::GetCurrentNode()` caches the result in a `thread_local int` on first call. Subsequent calls return the cached value without syscall.

Rationale: In QEMU, `sched_getcpu()` triggers a VM exit costing ~1000ns per call. Since Furrballs pins threads at initialization, the cached value is valid for the thread's lifetime. On real hardware, the overhead drops from ~1000ns to ~1-10ns (VDSO). The cache eliminates the dominant source of thread-local routing overhead in virtualized environments.

3.16 `FindAndEraseByHash` for Hash-Keyed Eviction

Decision: A dedicated `FindAndEraseByHash(HashPair)` operation accepts a pre-computed hash instead of a string key.

Rationale: ConcurrentARC's eviction path operates on `HashPair` values from ArcList (no original key string available). Without this operation, eviction would need to store original key strings or perform a reverse lookup. `FindAndEraseByHash` performs the same single-probe read+delete as `FindAndErase` but bypasses string hashing.

3.17 Placement-New Slot Initialization

Decision: After raw memory allocation, each slot is initialized via `new (&slots_[i]) Slot{}`.

Rationale: Raw memory from the allocator does not construct `std::atomic` members, leaving them in an indeterminate state. Reading an uninitialized `atomic<uint8_t>` is undefined behavior. Placement-new zero-initializes all members: seq starts at 0 (even = stable), fingerprint starts at 0.

3.18 `bool` Return on `replaceLocked/evictLocked`

Decision: `replaceLocked` and `evictLocked` return `bool` — false on CAS contention, callers propagate as ABANDONED_SET.

Rationale: These operations combine CMap mutation (CAS-based) with ARC list mutation. If the CMap CAS fails, the list must not be modified. The bool return enforces ordering: list mutation happens AFTER confirmed successful `FindAndEraseByHash`.

3.19 Unsigned Underflow Guard for `p_` Adjustment

Decision: `p_` decrement uses `p_ = (p_ >= dec) ? p_ - dec : 0;` instead of `std::max((size_t)0, p_ - dec)`.

Rationale: `p_` is `size_t` (unsigned). `p_ - dec` wraps to `SIZE_MAX` when `dec > p_`, and `std::max((size_t)0, SIZE_MAX)` returns `SIZE_MAX` (a no-op). The conditional guard prevents this silent underflow.

3.20 EvictionCallback Receives Only `const Value&`

Decision: The eviction callback signature is `std::function<void(const Value&)>`, not `std::function<void(const std::string&, const Value&)>`.

Rationale: The Value (KeyMeta) already contains all metadata needed for eviction: PageIndex, DataSize, DataOffset, NodeID. The key string was only used for ARC list tracking, which is now hash-keyed (§3.13). Passing the key string would require storing it or reconstructing it — unnecessary overhead.

3.21 `CMapSetResult` for Single-Probe INSERT/UPDATE Detection

Decision: `Set()` returns `CMapSetResult{Error err, bool inserted}` so callers know INSERT vs UPDATE from a single probe.

Rationale: Without this, ConcurrentARC::Set() would need to call `Find()` then `Set()` on CMap, probing the same key twice. The `inserted` flag enables ConcurrentARC to distinguish new entries (add to t1) from updates (no list change) with a single CMap operation.

3.22 Migration-Based Eviction over Per-Key Eviction

Decision: Cold keys are migrated from HOT pages to COLD pages rather than individually evicted from bump-allocated pages. Pages are evicted wholesale only when fully cold.

Rationale: The append-only bump allocator cannot reclaim per-key space without compaction (copy remaining values, update all KeyMetas). Per-key eviction leaves holes that accumulate under sustained eviction pressure. Migration avoids holes entirely: values are copied to receiving pages, and source page space is reclaimed when the page is emptied and recycled. The worst case is temporary memory overhead proportional to pages in transit, which is bounded and self-correcting.

3.23 Page Lifecycle: HOT/COLD/EMPTY/FREEZE

Decision: Pages cycle through four states: HOT (receives Set writes), COLD (receives migrated cold values), EMPTY (recyclable), FREEZE (mid-eviction, no writes allowed).

Rationale: The three productive states (HOT/COLD/EMPTY) cover all operational needs. FREEZE prevents the race where `Set()` writes to a page mid-eviction (new value would be lost when the page is flushed to persistent storage). `PageTier` is `std::atomic<PageTier>` to ensure `Set()` sees the FREEZE state without additional synchronization.

3.24 TempCtrl: Swiss Table-Style Temperature Tracking

Decision: Each page has a `TempCtrl` byte array parallel to `KeyIndex`, encoding each key's temperature (0x00 = HOT, 0x01 = COLD). The compactor uses SSE2 SIMD to scan for cold keys.

Rationale: This eliminates the need for the compactor to query ARC list membership (which requires the SpinLock). The eviction callback writes 0x01 under the existing SpinLock; the promotion callback writes 0x00 under the same SpinLock; the compactor reads without any lock (benign stale reads). SIMD scanning finds cold keys in O(keys/16) iterations. The byte array is inspired by Swiss table ctrl bytes (§2.7) and provides the same benefits: compact representation, SIMD-friendly, lock-free reads.

3.25 Hybrid Compaction Trigger

Decision: The compactor uses three complementary trigger mechanisms: ARC eviction callback sets `HasColdKeys` (free signal), background NodeJob worker(s) scan and migrate (amortizable), and cooperative migration on every 16th `Set()` on a flagged page (free labor under high write throughput).

Rationale: Pure ARC-triggered migration is expensive (memcpy + CMap ops on every eviction). Pure periodic scanning is fragile (interval tuning, cold keys linger). Pure pressure-based migration is

bursty (latency spikes under memory pressure). The hybrid model decouples the signal (ARC, near-free) from the work (compaction, deferred and batched) while allowing the Set path to contribute cooperatively under load.

3.26 Whole-Page Persistent Storage over Per-Key Writes

Decision: Evicted COLD pages are persisted as a single PageBlob (metadata + raw page data) rather than N individual RocksDB.Put() calls. Ghost hit recovery reloads the entire page speculatively.

Rationale: Single write per eviction amortizes RocksDB's LSM tree overhead (memtable, WAL, compaction) over the full page size (~4KB). Whole-page reload leverages the temporal locality of COLD pages: since the compactor migrates keys in ARC recency order, keys on a COLD page "went cold around the same time." Accessing one key often means others will follow soon (same principle as OS page fault handling). The PageBlob format preserves all per-key metadata (DataOffset, DataSize, Dirty) so reloaded pages have identical layout without recomputation.

3.27 Phase 2b Design Session: Decisions and Discarded Alternatives

This section documents the full decision-making process from the Phase 2b design session, including alternatives that were considered and discarded.

Eviction model

Decision	Chosen	Discarded
Eviction granularity	Migration-based: values move from HOT to COLD pages, COLD pages evicted wholesale	Per-key eviction from bump-allocated pages (leaves holes, requires compaction)
Page lifecycle states	HOT, COLD, EMPTY, FREEZE (4 states)	HOT, WARM, COLD pipeline (3 active states + EMPTY; rejected because HasColdKeys flag provides the same information with fewer state transitions)
FREEZE state	Included (prevents writes to page mid-eviction)	Not included initially; added as "free safety" with zero runtime cost
Value recovery from evicted pages	Whole-page speculative reload from RocksDB	Per-key RocksDB.Get() (more I/O operations, loses temporal locality); speculative pre-fetch on ghost hit (wastes I/O on keys never re-accessed)

Cross-node migration

Decision	Chosen	Discarded
CMap entry relocation	Move CMap entry from source to dest node (preserves sole-owner invariant)	Keep CMap entry on source, update KeyMeta to point to dest page data (breaks sole-owner invariant, concurrent Gets hit source node first anyway)
Migration ordering	Insert into dest CMap first, then erase from source (insert-before-erase)	Erase from source first, then insert into dest (creates transient miss window where key exists in no CMap, returning wrong KEY_NOT_FOUND)
Transient miss mitigation	Insert-before-erase eliminates the window entirely	MIGRATING ctrl byte forwarding tombstone (correct but unnecessary complexity); `std::unordered_map<uint64_t, uint8_t>` inflight migration table (correct but extra data structure for a solved problem)
Defer cross-node migration	Include in Phase 2b design	Defer to later phase (rejected because cross-domain migration is the thesis's topology-aware placement contribution)

Compactor trigger

Decision	Chosen	Discarded
Trigger model	Hybrid: ARC sets HasColdKeys flag, background NodeJob worker scans, cooperative Set() migration every 16th call	Pure ARC-triggered (expensive: memcpy + CMap ops on every eviction); pure periodic (fragile: interval tuning, cold keys linger); pure pressure-based (bursty: latency spikes under memory pressure)
Compactor execution	NodeJob worker(s) between dispatches	Dedicated thread per node (extra thread); global compactor thread (serializes across nodes)

Decision	Chosen	Discarded
Re-promoted key handling	TempCtrl byte set back to HOT by promotion callback	IsHot check during compaction via ARC SpinLock try_lock (adds SpinLock contention); separate promotion callback removing from ColdKeys vector (O(n) erase, concurrent modification risk)

Per-page tracking structure

Decision	Chosen	Discarded
TempCtrl location	Integrated in Page struct alongside KeyIndex	Extract into separate PageCompactInfo class (rejected because cooperative Set() migration and future mechanisms benefit from direct access without indirection)
TempCtrl lookup mechanism	To be determined (design parked for implementation phase)	KeySlotMap: `std::unordered_map<uint64_t, size_t>` per page (O(1) lookup, extra memory); Hash-to-position: Swiss-style hash of H2 to array index (zero extra memory, collision-tolerant); Linear scan of KeyIndex (simplest, acceptable for small pages)
KeyIndex/TempCtrl synchronization	Per-page CompactLock (SpinLock)	No synchronization (UB: std::vector concurrent modification); lock-free structure (unnecessary complexity for background operation)
Lock ordering	CompactLock always taken after ConcurrentARC's SpinLock (never before)	Not specified initially; added after identifying potential deadlock between eviction callback (ARC SpinLock → CompactLock) and compactor (CompactLock → no ARC SpinLock needed)

Persistence

Decision	Chosen	Discarded
Persistence layer	Keep RocksDB for Phase 2b, switch to mmap or custom format if measured as worse	Drop RocksDB entirely for mmap files (cleaner for whole-page blobs but loses crash safety and automatic compaction); file-per-evicted-page (cleanest I/O model but file system overhead at scale)
RocksDB key format	H2 (64-bit hash) as RocksDB key, or full 128-bit hash for zero collision risk	Store original key string in RocksDB (requires external key→string map, memory overhead)
Evicted page cleanup	TTL per page, shutdown compaction, or separate process/tool	None specified (unbounded RocksDB growth); in-memory only with process restart = cold cache (simplest but loses eviction state)

KeyMeta additions

Decision	Chosen	Discarded
CrossNodeAccesses storage	`std::atomic<uint32_t>` inside KeyMeta (Option A)	External `std::unordered_map<HashPair, atomic<uint32_t>>` (Option B: clean separation, extra lookup); plain `uint32_t` with benign races (Option C: heuristic counter, but atomic is zero-overhead on x86-64 and avoids UB)
Dirty flag	`bool` in KeyMeta (non-atomic, seqlock-protected)	Atomic bool (unnecessary: only written under CMap's UpdateInPlace seqlock)

Concurrency corrections (identified by review)

Issue	Fix	Severity
Compactor data-loss race: memcpy outside UpdateInPlace allows concurrent Set() to overwrite source data between memcpy and metadata update	Move memcpy inside the UpdateInPlace callback so read + metadata update are atomic under seqlock	Critical
std::vector concurrent modification: compactor and cooperative Set() both modify KeyIndex/TempCtrl without synchronization (UB)	Per-page CompactLock (SpinLock) for all KeyIndex/TempCtrl mutations, consistent lock ordering with ARC SpinLock	Critical
Eviction callback signature `void(const Value&)` lacks h2 needed for TempCtrl index lookup	Change to `void(uint64_t h2, const Value&)`	High
SIMD overread: final batch reads past TempCtrl.size(), causing out-of-bounds KeyIndex access	Mask final batch with `(1 << remaining) - 1`	Medium
Compactor processes FREEZE pages: concurrent compaction + page flush corrupts serialized data	Skip pages where `Tier == FREEZE` in compactor loop	Medium
Cross-node migration "acceptable for a cache" dismissal of transient miss window	Insert-before-erase ordering eliminates the miss window entirely; "it's a cache" is not a valid argument for returning wrong KEY_NOT_FOUND for keys that exist in memory	Critical

3.28 ARC Dimensionality: Recency is One Axis, Locality is Another

Context: The Phase 2b design (§2.9, §3.27) uses ConcurrentARC (key-level ARC, §2.8) as the eviction policy and bolts cross-node migration on top via TempCtrl state bits (REMOTE = 0x04)

and CrossNodeAccesses thresholds (§2.9.4). During design refinement, this approach produced persistent friction: "who sets REMOTE, when?" "What if a key is HOT but REMOTE?" "ARC says keep it, but locality says move it." The design session produced six categories of decisions (§3.27) but the cross-node migration category felt fundamentally ad hoc compared to the others. This section documents the insight that explains the friction and the architectural direction that resolves it.

The dimensionality mismatch

ARC classifies cache entries along exactly one dimension: **recency** (how recently was this entry accessed?). The ARC state machine has two inputs (access frequency, access recency) but produces one output (hot → cold → ghost → evicted). The `p_` tuning parameter adjusts where the boundary falls, but it's always a single boundary on a single axis.

Cross-domain migration requires classification along a **second dimension: memory locality** (is this key on the node that accesses it?). This produces four quadrants:

	LOCAL (correct node)	REMOTE (wrong node)
HOT (frequently accessed)	Ideal state. Do nothing.	Misplaced hot key. Urgent migration candidate.
COLD (rarely accessed)	Correct placement, low priority.	Bonus: evict or migrate lazily.
GHOST (recently evicted)	Normal ghost hit recovery.	Reload onto accessing node, not source node.

ARC's single-axis model cannot express these quadrants. When ARC says "this key is hot" (keep it), it has no opinion on whether the key is on the correct node. When ARC says "this key is cold" (evict it), it has no opinion on whether the key should be migrated rather than evicted. The Phase 2b design (§3.27) attempted to compress both dimensions into TempCtrl's 4-bit state field (HOT=0x1, COLD=0x2, REMOTE=0x4) but this creates ambiguous states: a HOT+REMOTE key (bitwise OR) is the most important migration candidate, yet TempCtrl's one-hot encoding cannot represent it. The design workaround was CrossNodeAccesses as a separate threshold counter, but this is orthogonal to ARC's recency tracking — it's a second policy pretending to be part of the first.

ARC's natural granularity is the page, not the key

ARC was designed (in the original Megiddo & Modha paper) as a block-level cache eviction policy for storage systems. The "blocks" in Furrballs are pages. ARC at the page level works cleanly because:

1. **Eviction is wholesale.** ARC says "evict page P." Dispatch to page P's node. The node handles it locally. No cross-node coordination needed for eviction itself.

2. **Location doesn't matter for eviction.** ARC doesn't care which node page P is on. It only cares about recency. The eviction decision is a single bit: "page P is cold, evict it."
3. **Ghost hit recovery is natural.** If a ghost page is accessed, reload it (§2.9.6). The page comes back as HOT. Clean state transition.

ARC at the key level (ConcurrentARC, §2.8) works for intra-node operations — hot keys stay on hot pages, cold keys migrate to cold pages within the same node. But key-level ARC has no mechanism to express "this key should move to a different node." That's not ARC's job. ARC tracks recency; it doesn't track locality.

What the Phase 2b design got right

The Phase 2b design (v1.8.0–v1.8.4) identified the correct mechanisms even if the framing was off:

- **TempCtrl as per-key hot/cold tracking** — correct. Each page needs to know which keys are hot and cold. This feeds the page-level ARC (cold pages are eviction candidates) and the compactor (migrate cold keys off hot pages).
- **Compactor** — correct. Cold keys on hot pages should be migrated to cold pages. This is an intra-node operation serving the page-level ARC.
- **Whole-page persistence** — correct. Cold pages evicted wholesale to RocksDB. This is page-level ARC's eviction mechanism.
- **Ghost hit recovery** — correct. Reload entire page from RocksDB. This is page-level ARC's promotion mechanism.
- **Insert-before-erase** — correct. The migration ordering (§3.27) eliminates transient miss windows. This applies to both intra-node compaction and cross-node migration.
- **CrossNodeAccesses in KeyMeta** — correct. Cross-node access counting is a Furrballs concern (not a CMap concern), thread-safe via atomic, and stored inside the seqlock-protected slot.

What needs to change: three separate mechanisms, three dimensions

The Phase 2b redesign separates concerns into three co-equal mechanisms, each handling one dimension:

1. Per-page TempCtrl (key-level hot/cold, intra-node): - Tracks which keys on each page are HOT or COLD. - Set by ARC eviction callback (key evicted from ARC → TempCtrl = COLD). - Set by ARC promotion callback (key promoted to ARC → TempCtrl = HOT). - Feeds the compactor: cold keys on hot pages → migrate to cold pages. - Feeds the page-level ARC: a page where all keys are COLD is a cold page candidate. - This is the ONLY connection between key-level ARC and the page lifecycle. TempCtrl is the bridge.

2. Page-level ARC (page-level hot/cold/ghost, cross-page eviction): - Tracks pages, not keys. Four ARC lists: hot pages (t1/t2), ghost pages (b1/b2). - Promoted by TempCtrl: when a page transitions from "has hot keys" to "all keys cold," it enters the cold page list. When a COLD page sits cold for N compaction cycles, it enters the ghost list. - Evicts wholesale: ghost pages

are evicted to RocksDB (§2.9.5). Dispatch to page's node. - Ghost hit recovery: reload page from RocksDB into HOT state (§2.9.6). - This is the "wholesale page eviction mechanism" that ARC was designed for. No key-level tracking, no per-key ghost lists, no b1/b2 confusion. Pages are the units.

3. Cross-node migration policy (key-level locality, cross-node): - Separate from ARC entirely. Not a recency policy; a locality policy. - Driven by CrossNodeAccesses threshold in KeyMeta (already designed, §3.27). - When threshold exceeded: migrate key to the accessing node (insert-before-erase, §2.9.4). - This is Furrballs' topology-aware placement contribution. It doesn't compete with ARC — it's orthogonal. A key can be HOT+REMOTE (ARC says keep, migration says move) or COLD+LOCAL (ARC says evict, migration says fine where it is). The two policies produce independent signals that are AND-ed, not OR-ed.

Implications for ConcurrentARC

Under this model, ConcurrentARC's role narrows:

- **CMap** — still needed. Concurrent key storage with seqlock reads. Topology-blind component. Unchanged.
- **ArcList** — still needed if key-level ARC tracking is preserved for intra-node compaction (ARC evicts key → callback sets TempCtrl = COLD → compactor migrates to cold page). But key-level ARC is now strictly an intra-node mechanism, not the eviction authority.
- **PromoteBuf** — still needed for deferred promotions out of the read path.
- **SpinLock** — still needed for ARC list mutations.
- **Key-level ARC lists (t1/t2/b1/b2)** — retained for intra-node hot/cold classification, but no longer the source of truth for eviction. Eviction authority moves to page-level ARC.

The key change: ConcurrentARC's eviction callback no longer triggers page eviction. It triggers TempCtrl state change (key → COLD). Page-level ARC reads TempCtrl aggregates to make page eviction decisions. This is a clean separation: key-level ARC classifies keys, page-level ARC classifies pages, cross-node migration classifies locality.

Open questions for the redesign

1. **Page-level ARC implementation:** Does the page-level ARC use the same ArcList structure (hash-keyed doubly-linked list) but with PageId instead of HashPair? Or is a simpler structure sufficient (pages are fewer than keys, iteration is acceptable)?
2. **Ghost page tracking:** When a cold page is evicted, it enters the ghost list. Ghost hit recovery (§2.9.6) promotes it back. Does the ghost list need the same two-list structure (b1 for recently evicted, b2 for twice-evicted)? Or is a single ghost list with LRU eviction sufficient?
3. **TempCtrl state count:** With ARC as an intra-node mechanism only, TempCtrl may no longer need the REMOTE state (0x04). Cross-node migration uses CrossNodeAccesses, not

TempCtrl, to flag migration candidates. TempCtrl becomes a pure ARC indicator: HOT (0x1) or COLD (0x2).

4. **Interaction between compaction and cross-node migration:** If the compactor migrates a key to a cold page (intra-node), and that key has high CrossNodeAccesses, should the compactor also initiate cross-node migration? Or does cross-node migration happen independently on its own schedule?
5. **Page-level ARC's p_ tuning:** Does the page-level ARC need adaptive `p_` like key-level ARC? With fewer pages than keys, the tuning dynamics may differ.

What survives from Phase 2b v1.8.4

Nearly everything survives. The mechanisms are correct; the framing is what changes:

Component	Survives?	Change
CMap	Yes	Unchanged
ConcurrentARC (key-level)	Partially	Retained for intra-node hot/cold classification. Eviction authority narrowed: ARC eviction callback → TempCtrl state change, not page eviction.
ArcList	Yes	Unchanged (key-level tracking)
PromoteBuf	Yes	Unchanged
TempCtrl	Partially	May drop REMOTE state. Becomes pure ARC indicator (HOT/COLD).
Compactor	Yes	Unchanged mechanism. Driven by TempCtrl (ARC's output), not ARC directly.
Insert-before-erase	Yes	Unchanged. Applies to both intra-node compaction and cross-node migration.
CrossNodeAccesses	Yes	Unchanged. Separate from ARC, separate from TempCtrl.
Whole-page persistence (PageBlob)	Yes	Unchanged. Driven by page-level ARC eviction.
Ghost hit recovery	Yes	Unchanged. Reload from RocksDB on ghost page access.
Page lifecycle (HOT/COLD/EMPTY/FREEZE)	Yes	Unchanged.
Page-level ARC (new)	**New**	ARC lists at page granularity. Drives wholesale page eviction.

Component	Survives?	Change
Cross-node migration policy (new framing)	**Reframed**	Same mechanism (CrossNodeAccesses + insert-before-erase), but explicitly a separate policy from ARC, not a bolt-on.

The Phase 2b design from §2.9 and §3.27 is not wrong — it's incomplete in one respect: it doesn't explicitly separate the recency axis (ARC) from the locality axis (Furrballs). The redesign adds this separation as an architectural principle, making the design cleaner and the cross-node migration story coherent. The code changes are minimal (add page-level ARC, possibly simplify TempCtrl). The conceptual change is significant.

3.29 REMARC: Reduction-Modeled Adaptive Replacement Cache

Context: The ARC dimensionality analysis (§3.28) established that recency and locality are orthogonal axes — ARC's single-axis model cannot express "this key is hot (keep it) but on the wrong node (move it)." The initial response was to bolt a separate locality policy (CrossNodeAccesses threshold) onto the existing ARC structure (§3.28's three-mechanism model). While architecturally correct, this approach stores recency, frequency, and locality as independent per-key metadata — three dimensions of tracking with no shared representation. This section documents the evolution from §3.28's three-mechanism model to REMARC, a unified policy that reduces all three dimensions into a single 2D state space, producing dual action scores via precomputed lookup tables.

On the name: REMARC stands at ARC's level as a peer policy, not an extension. It borrows ARC's adaptive principle (observe past decisions, adjust future behavior) but replaces ARC's deterministic list-based mechanism with a probabilistic smoothed state space. The acronym expands to **Reduction-Modeled Adaptive Replacement Cache**: Reduction (3D → 2D state space), Modeled (mathematical IIR smoothing, not heuristic), Adaptive (self-tuning thresholds via reversal feedback), Replacement (cache replacement policy), Cache (domain). The all-uppercase treatment signals it is a distinct policy, not a variant of ARC.

ARC territory: REMARC shares ARC's intellectual DNA — the principle of self-tuning via observed outcomes and the balancing of recency and frequency — but does not share ARC's mechanism. No t1/t2/b1/b2 lists, no ghost lists, no p_ parameter. These are general cache principles (LRU has recency, LFU has frequency), not ARC-specific inventions. REMARC is ARC-inspired but architecturally independent.

Design evolution (§3.28 → REMARC)

Step 1: Frequency as dual-purpose signal. ARC's frequency dimension (t2) tracks "how much" — a key is important if accessed frequently. Cross-node migration needs "from where" — which node accesses this key. Rather than tracking these separately, frequency can serve double duty: as a scalar (total access count → eviction decisions) and as a per-node vector (which node accesses most → migration target). This insight preserved ARC's frequency tracking while extending it to carry locality information.

Step 2: Probabilistic HotNode with migration disfavor. Per-node frequency counters for every key are expensive (N-1 counters per key). A probabilistic approach reduces this to a single `uint8_t HotNode` per key, updated with probability p on each cross-node access. By setting p low (biased against updating HotNode), migration is naturally disfavored — only sustained, repeated access from the same remote node overcomes the bias. The bias IS the anti-ping-pong mechanism; no separate ghost list or cooldown needed.

Step 3: Recency as soft anti-migration bias. Rather than a hard-coded bias parameter, the anti-migration force should be emergent from the local access pattern. If a key is accessed locally frequently, it resists migration strongly. If local access fades (stale), resistance softens naturally. This "leans with recency" — the same recency signal that ARC tracks for eviction also opposes migration. Recency and locality are not just orthogonal; they actively compete through a shared smoothing function.

Step 4: Unified state space (the reduction). The three input dimensions (recency, frequency, locality) can be reduced to two stored values per key: `S_local` (smoothed local access signal) and `S_remote` (smoothed remote access signal). From these two values, all three dimensions and both decision scores are derived:

```
R = S_local / MAX                                    // recency
F = (S_local + S_remote) / (2 × MAX)                 // frequency
L = S_remote / max(S_local + S_remote, 1)             // locality (remote fraction)
```

This is the dimensional reduction: 3 inputs → 2 stored values → 2 outputs. The intermediate representation (R, F, L) exists only as derived quantities, never stored.

Step 5: Decision surfaces. The two action scores are simple products of the derived dimensions:

```
Evict(S)   = (1 - R) × (1 - F)       // evict when cold AND infrequent
Migrate(S) = F × L × (1 - R)          // migrate when frequent AND remote AND locally stale
```

Furrballs evaluates both scores independently. The execution priority is migration-first: `if M > _m: migrate to HotNode`, then `if E_page > _e: evict page`. Migration is processed first because evicting a page discards all keys on it — if a high-M key (migration candidate) lives on a high-E_page page (eviction candidate), migrating the key first extracts it before the page is evicted. After migration, E_page is recomputed (the migrated key no longer contributes). If

E_page remains above _e, the page is evicted. This ordering preserves migration candidates that would otherwise be lost to page eviction.

Step 6: SIMD lookup tables. Storing `S_local[4bit]` and `S_remote[4bit]` in one TempCtrl byte per key, the byte value `(S_remote << 4 | S_local)` is a direct index into a 256-entry precomputed lookup table. For SIMD batch scoring, SSE2 nibble-unpack and 16-bit multiply arithmetic computes E and M scores for 16 keys per batch (~18 instructions). Note: `pshufb` cannot perform 256-entry lookups directly (4-bit index limitation), and the E/M formulas are not separable into independent nibble lookups, so direct SIMD arithmetic is used instead. Scanning a page of ~64 keys becomes 4 batches — effectively O(1) per page. The lookup tables are static (computed from the decision surface formula); only the thresholds (_e, _m) change at runtime.

Discarded approaches during evolution:

Approach	Why discarded
Three separate trackers (TempCtrl + ConcurrentARC + CrossNodeAccesses)	Three independent data structures for three dimensions. No dimensional reduction.
Per-node frequency counters per key	Expensive: N-1 × 4 bytes per key. For 8 nodes: 28 bytes. Replaced by probabilistic HotNode (1 byte).
Fixed migration bias parameter	Doesn't adapt to workload changes. A key that was migration-worthy at t=0 might not be at t=1000.
Hard EMA smoothing factor (fixed)	Doesn't capture the insight that recency should actively oppose migration. Replaced by separate local/remote smoothing with asymmetric decay.
Full 2D ARC with 8 lists (t1, t2, t1r, t2r, b1, b2, b1r, b2r)	Two independent ARC structures (one for recency, one for locality) with a combined 8-list management. Complex, and the probabilistic smoothed approach produces equivalent decisions with less code and memory.
Fixed-p adaptation (standard ARC)	Requires ghost lists to observe eviction history. REMARC's outcome-based reversal feedback achieves the same self-tuning without maintaining ghost state.

Formal definition

State vector per key:

```
S = (S_local, S_remote)   [0, 15]²     (4-bit encoding in TempCtrl byte)
Plus: uint8_t HotNode                     (probabilistically updated, stored in KeyMeta)
```

Update rules (first-order IIR smoothing):

The update model uses IIR smoothing with cross-channel erosion. S_local and S_remote represent smoothed estimates of the local and remote access rates. Observing an access of one type is evidence *against* the other type occurring in the same time window — so the non-observed rate decays. This mutual erosion captures the intuition that local and remote access rates compete: observing one is evidence against the other.

Quantization note: All operations use 4-bit integer arithmetic (values 0–15). Integer division introduces systematic bias: truncation causes downward drift (scores converge to 0 faster than the continuous formula predicts). The implementation uses round-to-nearest (add half the divisor before dividing) to eliminate this bias. The lookup tables are unaffected — they compute exact scores from the quantized inputs.

```
On local Get():
  S_local  += round( _l × (15 - S_local) / 15)     // boost local, round to nearest
  S_remote  = round(S_remote × (15 -  _r) / 15)     // decay remote, round to nearest

On remote Get() from node N:
  S_remote += round( _r × (15 - S_remote) / 15)     // boost remote, round to nearest
  S_local   = round(S_local × (15 -  _l) / 15)      // decay local, round to nearest
  with probability p(S_remote): HotNode = N            // probabilistic migration target

Time decay (per scan cycle):
  S_local  = round(S_local ×  _num /  _den)           // both drift toward 0
  S_remote = round(S_remote ×  _num /  _den)
```

Parameters _l, _r, are fixed (control EMA responsiveness). The asymmetry (1- _l) vs (1- _r) determines migration bias: _l > _r means local signal recovers faster than remote signal, disfavoring migration.

Derived dimensions:

```
R = S_local / 15                            // recency (normalized local signal)
F = (S_local + S_remote) / 30              // frequency (normalized total signal)
L = S_remote / max(S_local + S_remote, 1)  // locality (remote fraction)
```

Decision surfaces:

```
Evict(S)   = (1 - R) × (1 - F)            // high when key is cold and infrequent
Migrate(S) = F × L × (1 - R)               // high when key is frequent, remote, and locally st
```

Quadrant partition of the (S_local, S_remote) state space:

```
              S_remote →
              low              high
S_local
high       KEEP                        CONTESTED
           E   0, M   0                E   0, M moderate
           hot local key                both active,
                                        R resists migration

low        EVICT                       MIGRATE
           E   high, M   0             E   low, M   high
           cold, not remote             cold locally,
                                        hot remotely = misplaced
```

SIMD-optimized scan:

```
// Precomputed once (static):
uint8_t E_lookup[256];   // byte = (S_remote << 4 | S_local) → Evict score
uint8_t M_lookup[256];   // byte = (S_remote << 4 | S_local) → Migrate score

// Per batch of 16 keys:
__m128i tc = _mm_loadu_si128(page.TempCtrl + batch);
__m128i s0 = _mm_and_si128(tc, _mm_set1_epi8(0x0F));   // extract S_local
__m128i s1 = _mm_srli_epi16(tc, 4);                      // extract S_remote
// Compute E and M via SIMD 16-bit arithmetic (~18 instructions total)
// (direct lookup via pshufb not possible - 256-entry table, 4-bit index limit)
__m128i e = /* SIMD compute Evict(s0, s1) */;
__m128i m = /* SIMD compute Migrate(s0, s1) */;
__m128i e_mask = _mm_cmpgt_epi8(e,  _e_vec);       // evict candidates
__m128i m_mask = _mm_cmpgt_epi8(m,  _m_vec);       // migrate candidates
```

Adaptation (outcome-based, ghost-less):

Parameter	Adaptation signal	Mechanism
_e (eviction threshold)	evict_reload_rate	If keys evicted then reloaded from RocksDB within K cycles > target → _e was too low → increase. If miss rate high AND reload rate low → decrease.

Parameter	Adaptation signal	Mechanism
_m (migration threshold)	migrate_reversal_rate	If keys migrated then accessed from source node within K cycles > target → _m was too low → increase. If cross-node latency high AND reversal low → decrease.

This is ARC's ghost feedback principle (observe wrong decisions, adjust) applied without maintaining ghost lists. The feedback signal is the same; the observation mechanism is outcome-based rather than state-based.

Implications for Phase 2b design

REMARC is a new design direction (Direction A from §3.28). Direction B (ConcurrentARC for eviction + separate migration policy) remains documented as an alternative. The two directions are not mutually exclusive — REMARC could replace ConcurrentARC's key-level policy while ConcurrentARC's CMap backing remains unchanged (CMap provides concurrent storage, REMARC provides the policy).

Component	Direction A (REMARC)	Direction B (complement ARC)
CMap	Unchanged	Unchanged
ConcurrentARC lists (t1/t2/b1/b2)	Replaced by REMARC's (S_local, S_remote)	Retained for eviction
ArcList	Not needed (no list membership)	Retained
PromoteBuf	Not needed (no deferred promotions)	Retained
TempCtrl	Reencoded: (S_local[4bit], S_remote[4bit])	Retained as flags (HOT/COLD/REMOTE_CANDIDATE)
HotNode (KeyMeta)	New: 1 byte, probabilistic	New: equivalent concept
SIMD scan	Precomputed lookup tables (SSE2 arithmetic)	Flag scanning (cmpeq)
Adaptation	Outcome-based reversal feedback	ARC ghost list p_ adjustment
Memory per key	3 bytes (TempCtrl 1 + HotNode 1 + eviction/migration derived)	4+ bytes (TempCtrl 1 + ArcList 16 + CrossNodeAccesses 4)

Page-level ARC collapse and one-scanner unification

The aggregation insight: REMARC's per-key Evict scores can be aggregated per page to drive wholesale page eviction, eliminating the need for a separate page-level ARC entirely. The page's eviction score is the average of its keys' Evict scores:

```
E_page = (1/N) × Σ E_key = (1/N) × Σ (1 - R_i) × (1 - F_i)
```

- Page with all hot keys: E_page 0 → keep
- Page with all cold keys: E_page 1 → evict to RocksDB
- Page with mixed keys: E_page 0.5 → borderline

No separate page-level ARC lists (t1_page/t2_page/b1_page/b2_page), no per-page SpinLock, no TempCtrl-to-page-ARC bridge mechanism. The "wholesale page eviction" that was the original ARC intent (§3.28) becomes an emergent property of REMARC's key-level scoring.

The SIMD scanner computes E_page for free:

```
// Same scanner, same infrastructure, now with per-page aggregation:
__m128i tc = _mm_loadu_si128(page.TempCtrl + batch);
__m128i s0 = _mm_and_si128(tc, _mm_set1_epi8(0x0F));
__m128i s1 = _mm_srli_epi16(tc, 4);
__m128i e  = /* SIMD compute Evict(s0, s1) */;    // 16 Evict scores
__m128i m  = /* SIMD compute Migrate(s0, s1) */;   // 16 Migrate scores
// Horizontal sum of e → accumulate into page's E_page
```

One scanner, two thresholds: The compactor (intra-node), evictor (page-level), and migration scanner (cross-node) become the SAME scanner operating on the same scores:

Scanner role	Score	Threshold	Action
Migration	Migrate(S)	_m	High M key → migrate to HotNode
Compaction	Evict(S) per key	_compact	High E key → move to cold page
Page eviction	E_page (aggregated)	_e_page	High E_page → evict page to RocksDB

The three-mechanism model from §3.28 (TempCtrl + page-level ARC + cross-node migration policy) collapses to ONE mechanism: REMARC. One TempCtrl encoding, one SIMD scanner, two action scores.

Ghost-less reload protection: When a page is evicted to RocksDB and reloaded on access, the keys' REMARC scores are lost. If keys start at (S_local=0, S_remote=0), then E = (1-0)×(1-0)

= 1 → immediately re-evictable. This is the classic ghost-hit problem that ARC solves with ghost lists.

REMARC solves it without ghost lists: on page reload, set S_local = MAX for all keys. This means E = (1-1)×(1-F) = 0 → protected from immediate re-eviction. The key appears as "recently accessed locally" — which is TRUE, since the reload was triggered by an access. Natural decay via gradually makes the key evictable again, but only after sustained coldness. This IS ARC's ghost hit promotion (back to t1/t2) without ghost lists: the "promotion" is just setting the initial score on reload.

This eliminates the last piece of ARC's machinery that might have been needed. REMARC is fully self-contained: no ghost lists, no per-page ARC, no separate migration policy. One policy, one TempCtrl encoding, one scanner.

Open questions

1. **4-bit vs 8-bit encoding:** 4 bits per score (16 levels) is compact (1 byte per key) but coarse. 8 bits (256 levels) requires 2 bytes per key and SSE/AVX for arithmetic, but provides better precision for EMA smoothing. Empirical comparison needed.
2. **__l, __r, calibration:** These control EMA responsiveness. Should be empirically determined or derived from workload characteristics (e.g., total key count, access rate).
3. **__e, __m initial values and step sizes:** The adaptation loop needs starting points and step sizes. Too aggressive adaptation causes oscillation; too conservative causes slow convergence.
4. **HotNode probability function p(S_remote):** Should p increase linearly with S_remote (more confident with more remote signal), or is a fixed p sufficient? Linear p provides faster convergence for high-traffic keys.
5. **E_page aggregation strategy:** Compute per-scan via SIMD horizontal sum (O(1) amortized), or maintain a running aggregate updated on each access (O(1) per access but shared mutable state under contention)?
6. **Migration execution:** Insert-before-erase (§3.27) remains the mechanism. Does the migration trigger happen synchronously (on Get() when M > _m) or asynchronously (scanner detects, background worker executes)?
7. **Reload S_local initialization:** Set to MAX (full protection, slowest convergence to evictable) or a fraction (faster convergence but less ghost protection)? Related to the ghost-less reload design (§3.29).
8. **Adam-style adaptive smoothing (future optimization):** Instead of fixed _l, _r, track a second moment (variance) per score: `V =  × V + (1- ) × (ΔS)²`, then adapt `= _base / (√V + )`. Keys with stable access patterns converge faster; keys with volatile patterns converge slower — resisting premature migration from noisy bursts. Adds ~2 bytes per key for variance tracking. Noted for post-validation optimization if fixed proves insufficient.
9. **ML parallels for thesis framing:** REMARC's update rule is mathematically equivalent to gradient descent (`S +=  × (target - S)`, where (target-S) is the gradient), with L2 regularization via time decay (`S *= `). The probabilistic HotNode sampling is -greedy

exploration. The adaptation loop is online learning with outcome feedback. These analogies strengthen the theoretical grounding but do not change the implementation.

3.30 REMARC Algorithm Paper and Policy Framework

During Phase 2b development, the REMARC algorithm evolved from the initial 2D formulation (§3.29) into a general policy framework documented in a companion paper [Sphynx2025]. The framework formalizes cache policies as compositions:

```
E = P(A , ..., A )
```

where `A_i` are **atoms** (access signal extractors like recency, frequency, period) and `P` is a **projection** that maps atom values to a discrimination function for cache decisions. REMARC's original formulation is the specific instance `E = P(recency, frequency)` with a linear projection and integer quantized state.

ARC as a special case. The REMARC framework subsumes ARC as a strict subset. The augmentation investigation [Sphynx2025, §8] demonstrates that adding REMARC's richer state (inter-arrival period, confidence) to ARC's quota structure produces AUG-TS4 (timestamp-based demotion), which matches or exceeds ARC on three of four workloads. Setting the demotion threshold to minimum reproduces ARC exactly (the strict subset property).

Variant taxonomy. 24 policy variants were evaluated in PolicyBench, classified by projection mechanism:

Projection class	Variants	Key idea
Score (continuous)	REMARC, OPT, LazyInc, LFU, TinyLFU	P maps atoms to scalar eviction score
Gate (binary)	S3, BloomGate, GhostGate	P returns admit/reject decision
Modulate	BloomMod	P adjusts initial state
Factor	FACT, Dual, MS	P decomposes into sub-projections
Step (threshold)	STEP, STEPK	P maps to discrete eviction set
Augment (ARC + state)	TS, PRED, HEAP, BOTHEND, ADAPT	ARC quota + REMARC delivery
Quota-free (no ARC)	QuotaFree	Pure predictive eviction, no structural protection

Key findings relevant to Furrballs:

- ARC achieves 35.17% Zipfian hit rate at 10% capacity. Page-batch REMARC variants are bounded at ~24.5% (Finding 1). This ceiling is structural: page-level eviction discards per-key scoring.
- AUG-ADAPT, a self-tuning variant with adaptive delivery selection, achieves 99.33% on adversarial looping workloads while matching ARC on Zipfian/ScanRes/Temporal (Finding 30). It uses a feedback controller (population CV + prediction regret) to switch between heap-based and LRU-based eviction.
- The delivery–quota tension (Finding 27): more accurate eviction selection requires bypassing ARC's list structure, but this corrupts ARC's p-adaptation. AUG-ADAPT resolves this via self-tuning rather than architectural change.
- QuotaFree (pure prediction, no ARC structure) collapses on scan workloads (11.38% Zipfian, 12.01% ScanRes vs ARC's 35.17%/88.90%) because cold keys with no access history cannot be predicted (Finding 31). Ghost lists provide delayed future observation that pure prediction cannot replicate.

The companion paper [Sphynx2025] provides the full evaluation, theory, and algorithmic details. Furrballs serves as the implementation platform for real-hardware deployment on NUMA and other asymmetric memory topologies.

3.31 Engineering Decisions (Phase 2b Development)

The PolicyBench simulation harness and REMARC paper evaluation drove several engineering decisions that affect Furrballs' future policy integration:

Policy as a compile-time type parameter. Furrballs uses `FurrBall<Policy>` where Policy is a template type (§2.8). This was extended to support REMARC variants during PolicyBench development. All 24 variants share a common `access(uint64_t key)` interface with `hits()`, `misses()`, `evictions()`, and `scans()` queries. No virtual dispatch, no runtime polymorphism. This design validated cleanly — adding variants required only template instantiation, no interface changes.

Three hash structures evaluated. The development progressed through three data structure strategies: 1. **Triple-hash (AUG-TS):** Separate `unordered_set` (cached keys), `ArcList` (list position), and `unordered_map` (REMARC state). Three hash lookups per access. Simple but slow (3.5M ops/s). 2. **Merged-hash (AUG-PRED):** Single `unordered_map<uint64_t, Entry>` combining all three. One lookup per access. 4.5M ops/s. The merged structure demonstrated that performance is a data structure problem, not an algorithmic one. 3. **Heap-augmented (AUG-HEAP):** Merged hash plus `priority_queue` for eviction selection. Lazy deletion via version stamps. 7.0M ops/s on Zipfian (faster than ARC's 5.4M due to lower per-access overhead).

Ghost list management (Finding 18). A critical bug was discovered during augmentation: ARC's `doEvict()` must DROP evicted keys, not move them to ghost lists. Moving evicted keys to ghost lists inflated the ghost lists, causing excessive p-adaptation and catastrophic Zipfian regression (17.82%). Fixing this bug produced byte-for-byte identical results to ARC (105,508 hits on identical

trace), confirming that the implementation is correct. The bug was identified by tracing key 0 through both ARC and AUG side-by-side, finding the first divergence at operation 4000, and tracing back to the ghost management difference.

Timestamp-based decay (Finding 20). Lazy per-key decay (run only on access) cannot demote idle keys because idle keys have frozen state. The frozen state problem blocks demotion for keys no longer being accessed. This motivated the timestamp-based decay approach (compute idle time on re-access), which enables evidence-based demotion.

Feedback controller for adaptive delivery (Finding 30). AUG-ADAPT combines population-level workload characterization (CV of inter-arrival gaps) with per-eviction prediction regret to self-tune between LRU and heap-based eviction. The controller uses two signals: feedforward (population CV discriminates homogeneous from heterogeneous patterns) and feedback (regret from ghost hits measures prediction accuracy). The control law is `confidence = (1 - 3*CV) * (1 - regretEMA)`, with heap eviction when confidence > 0.5. This is a proper control loop with the workload as the plant, the delivery mechanism as the controller, and hit rate as the setpoint.

4. Related Work

System	Memory Topology Approach	Difference from Furrballs
TCMalloc	Topology-aware allocation	Allocator only; no cache eviction or page management
jemalloc	Topology-aware arenas	Allocator only; no cache policy
CacheLib (Meta)	Static sharding per node	Coarse-grained; entire cache instance per node
PostgreSQL	Shared buffers with partition locks	Database buffer pool; not a standalone cache
InnoDB	Buffer pool instances	Database storage; not a cache library
Redis	No topology awareness	Baseline comparison point
Rayhan & Aref (2025)	Custom `move_pages2` kernel syscall	Kernel-level page migration optimization for DBMS; no cache placement policy
US11561834B2 (Rambus)	ML-based metablock placement via RL	External optimizer using reinforcement learning; no per-page cache eviction
Sphynx (2025) [Sphynx2025]	General cache policy framework (REMARC)	Algorithm-level policy framework; no topology awareness, no systems implementation

System	Memory Topology Approach	Difference from Furrballs
Furrballs	Per-page placement + REMARC unified policy	Fine-grained, adaptive, page-level memory locality (NUMA in current implementation) as policy input

To our knowledge, no published system implements page-level cache eviction policy informed by asymmetric memory placement and cross-domain access frequency. Topology-aware allocators (TCMalloc, jemalloc) handle placement but not eviction; topology-aware caches (CacheLib) use static sharding without adaptive per-page locality decisions. Rayhan & Aref's `move_pages2` syscall improves kernel-level page migration efficiency for DBMS workloads (up to 1.84x query throughput improvement) but does not address user-space cache placement policy—it is complementary to Furrballs' approach, which avoids runtime page migration entirely via track-and-defer eviction. Rambus's US11561834B2 patent describes an ML-based reinforcement learning system for adaptive metablock placement across NUMA tiers, operating as an external optimizer above the OS rather than as a user-space caching library with integrated topology-aware eviction. The companion REMARC paper [Sphynx2025] (DOI: 10.5281/zenodo.19794758) formalizes the policy framework and evaluates 24 variants across five workloads; Furrballs provides the systems platform for topology-aware deployment of that framework.

5. Evaluation

5.1 Methodology

Benchmarks run on three platforms: - **Host (baseline):** Single NUMA node, VPS, Ubuntu 24.04, x86_64, GCC 14, CMake Release - **QEMU VM (NUMA simulation):** 2 NUMA nodes, 4 vCPUs (2 per node), 4GB RAM, Ubuntu 24.04 - **AWS EC2 metal (real hardware):** c6i.metal (Intel Ice Lake, 2 NUMA nodes, §5.18) and c6a.metal (AMD Milan, 4 NUMA nodes, §5.26)

Each scenario is repeated 3–10 times depending on section (QEMU sections: 3–5 runs; real hardware sections: 10 runs). Reported values are averages across iterations. The QEMU VM adds approximately 600ns of emulation overhead per operation. To isolate the asymmetric memory effect (NUMA in this case), we compare local vs cross-domain reads under identical conditions within the VM.

Thread placement: Multi-threaded benchmarks pin each worker thread to a specific NUMA node via `PinCurrentThreadToNode()` (wrapping `numa_run_on_node()`). This is strictly a measurement concession to isolate local vs cross-node latencies without scheduler migration confounds. It is not part of the library's contract — Furrballs treats memory topology as a first-class internal concern: per-domain allocation, key routing, and eviction are all handled by the library without requiring callers to be topology-aware or to pin threads. The internal `NodeJob` workers are the only

permanently pinned threads. In production, callers run unmodified; the library detects and adapts to the topology at initialization.

Simulated NUMA latency: Since QEMU does not simulate real cross-node memory access latency, we optionally inject a calibrated busy-wait delay via the compile flag `-DSIMULATE_NUMA_LATENCY_NS=VALUE`. When enabled, `Get()` detects whether the calling thread is on a different NUMA node than the target key's placement node, and spins for `VALUE` nanoseconds using `rdtsc`-based busy-wait with runtime TSC calibration. The delay is placed before the `SeqLock::Read()` call to model the cost of accessing remote memory. Default values are sourced from published hardware specifications: Intel Xeon cross-socket ~70ns, AMD EPYC Milan ~50ns. This simulation validates that the architecture responds correctly to NUMA latency signals, but does not substitute for real hardware measurement.

Build command for simulated benchmarks:

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DSIMULATE_NUMA_LATENCY_NS=70
cmake --build . --target Benchmark
```

5.2 Single-Threaded Baseline (Host, 1 NUMA Node, 5 iterations averaged)

Value Size	Operation	Throughput (ops/s)	p50 (ns)	p99 (ns)	Stddev (ns)
64B	Set	4,333,091	146	830	1,060
64B	Get	4,473,004	134	255	1,394
512B	Set	3,753,818	171	1,433	1,332
512B	Get	4,735,494	147	245	193
4KB	Set	1,660,059	926	2,686	1,167
4KB	Get	2,888,215	270	442	241

5.3 Single-Threaded (QEMU VM, 2 NUMA Nodes, 5 iterations averaged)

Value Size	Operation	Throughput (ops/s)	p50 (ns)	p99 (ns)	Stddev (ns)
64B	Set	679,054	695	2,674	3,416
64B	Get	766,818	655	915	820
512B	Set	581,262	760	3,249	7,293
512B	Get	757,711	672	985	918
4KB	Set	499,495	1,561	6,170	2,170
4KB	Get	677,780	799	1,048	1,256

Host-to-VM ratio: approximately 5-6x slower in the VM due to QEMU emulation overhead.

5.4 Cross-Domain Effect (NUMA Instance) (QEMU VM, 2 threads, 5 iterations averaged)

Two threads, each pinned to a different NUMA node. One thread writes keys to its local node, the other reads them from the remote node. Results compare shared_mutex reads (v0.5) vs SeqLock lock-free reads (v0.6).

Value Size	Metric	Local (ns)	Cross-Node (ns)	Overhead
64B (v0.5, shared_mutex)	p50	956	1,005	5.1%
64B (v0.5, shared_mutex)	p99	1,869	2,075	11.0%
64B (v0.6, SeqLock)	p50	1,157	1,292	**11.7%**
64B (v0.6, SeqLock)	p99	2,093	2,372	**13.3%**
512B (v0.6, SeqLock)	p50	1,252	1,299	3.8%
512B (v0.6, SeqLock)	p99	2,199	2,243	2.0%

Key finding: Lock-free reads (SeqLock) expose a cross-node p50 overhead of 11.7%, more than double the 5.1% visible under `shared_mutex`. The lock was masking the true cross-domain memory latency signal. The p99 overhead increased from 11.0% to 13.3%, confirming the effect is consistent across latency percentiles. The 512-byte results remain noisy due to QEMU scheduling artifacts—real hardware validation is expected to show a cleaner signal at all value sizes.

This result validates the thesis methodology: synchronization overhead must be eliminated from the read path to measure the true cross-domain effect.

5.5 Concurrent Throughput (QEMU VM, 4 threads, 5 iterations averaged)

Four threads, each pinned to a NUMA node, operating on disjoint key ranges. With SeqLock lock-free reads, cross-node Get throughput exceeds local.

Value Size	Mode	Set (ops/s)	Get (ops/s)
64B	All local (2 threads/node)	761,315	1,197,488
64B	All cross-node	952,933	1,592,695

Cross-node Get throughput is 33% higher than local under lock-free reads. This is **not** a NUMA topology effect—QEMU does not simulate real cross-node memory latency (all memory accesses have identical latency within the VM). The anomaly likely reflects QEMU's vCPU scheduling behavior: when threads access memory on the "remote" node, the hypervisor may schedule the accessing vCPU on the core closest to that memory region, reducing scheduling contention. On real hardware, this result would invert: cross-node Gets would be slower due to genuine remote memory latency (typically 30-100ns additional overhead on multi-socket systems). This section is retained as evidence that QEMU NUMA simulation does not reproduce hardware NUMA effects, reinforcing the need for real hardware validation.

5.6 Routing Strategy Comparison (QEMU VM, 2 threads, 5 iterations averaged)

Two threads, each pinned to a different NUMA node (node 0 and node 1). Each thread writes 5000 keys (64 bytes), then reads its own keys (self-read) and the other thread's keys (cross-read). Results compare shared_mutex (v0.5) vs SeqLock lock-free reads (v0.6).

v0.5 (shared_mutex reads):

Metric	Round-robin (ns)	Thread-local (ns)	Improvement
Set p50	1,033	913	**11.6%**
Set p99	11,288	5,503	**51.2%**
Self-Get p50	907	835	**8.0%**
Self-Get p99	2,096	1,357	**35.3%**
Cross-Get p50	879	792	**9.9%**
Cross-Get p99	1,953	1,358	**30.5%**

v0.6 (SeqLock lock-free reads):

Metric	Round-robin (ns)	Thread-local (ns)	Improvement
Set p50	1,621	1,355	**16.4%**
Set p99	41,095	4,665	**88.6%**
Self-Get p50	1,296	955	**26.3%**
Self-Get p99	2,600	1,790	**31.2%**
Cross-Get p50	1,292	901	**30.3%**
Cross-Get p99	2,677	1,582	**40.9%**

Lock-free reads amplify the thread-local routing improvement from 8-10% p50 to **26-30% p50**. The lock was masking not only the cross-domain effect but also the routing strategy benefit. Thread-local routing with SeqLock achieves 26-41% improvement across all Get metrics, confirming that the compound benefit of topology-aware placement (memory locality + reduced synchronization) is genuine and measurable.

5.7 Baseline Comparison: Cross-VM Isolation (3 runs each)

To isolate the topology contribution, Furrballs runs in the NUMA VM (2 nodes) and the BaselineCache runs in a separate non-NUMA VM (1 node, same QEMU overhead, no NUMA topology). The BaselineCache uses identical mechanisms (SeqLock reads, CAS bump allocator, per-op timing) but eliminates all topology-specific design choices: single `malloc`-backed allocation, single `unordered_map` (no per-node shards), single `shared_mutex`.

Single-threaded 64B (3 runs averaged):

Implementation	Set p50 (ns)	Get p50 (ns)	Set ops/s	Get ops/s
Furrballs (NUMA VM, round-robin routing)	825	786	629K	679K
Baseline (non-NUMA VM)	802	671	582K	667K
Delta	+2.9%	+17.1%	+8.1%	+1.8%

Concurrent 4-thread 64B (3 runs averaged, both with per-op timing):

Implementation	Set ops/s	Get ops/s
Furrballs (NUMA VM, round-robin routing)	1,240K	1,630K
Baseline (non-NUMA VM)	383K	1,356K
Ratio	**3.24x**	**1.20x**

Analysis:

Furrballs' concurrent throughput *exceeds* the baseline in both Set (3x) and Get (1.2x). The mechanism is lock partitioning: Furrballs' per-node sharding creates N separate `shared_mutex` instances (one per NUMA node), so 4 threads contend on 2 mutexes instead of 1. Round-robin routing distributes keys evenly across shards, ensuring balanced contention. The baseline's single `shared_mutex` becomes the bottleneck under concurrent writes—all 4 threads serialize on one lock.

Single-threaded overhead is mixed: Set p50 is +2.9% (within noise), but Get p50 is +17.1%, reflecting the cost of shard iteration when the key's location is unknown. The Get overhead is the architectural cost of per-node isolation.

Caveat: QEMU does not simulate real cross-node memory latency. Remote memory access within the VM has identical latency to local access. On real hardware, the NUMA penalty (1.2-2x remote access overhead) would partially offset the lock partitioning benefit for read operations. The cross-VM comparison captures the architectural benefit (lock partitioning, smaller per-shard maps) without the hardware NUMA cost. Real hardware validation is needed to measure the true net effect.

Initial in-VM comparisons showed a 6-8x concurrent Get gap, which was traced to a benchmark methodology issue: Furrballs measured per-operation latency via `Clock::now()` (two VM exits per operation in QEMU), while the baseline measured only total wall time. With identical per-op timing and separate VMs, the gap reversed to a 1.2x advantage for Furrballs.

5.8 Simulated NUMA Latency (70ns, QEMU VM, single run)

With `-DSIMULATE_NUMA_LATENCY_NS=70` (modeling Intel Xeon cross-socket latency), a busy-wait spin injects 70ns before each cross-node `SeqLock::Read()`. Under thread-local routing, self-Get

operations access the local node (no delay), while cross-Get operations access the remote node (70ns delay).

Single-threaded 64B:

Configuration	Get p50 (ns)	Get ops/s
No simulation	720	718K
70ns simulation	1,025	565K
Delta	+42.4%	-21.3%

Concurrent 4-thread 64B:

Configuration	Get ops/s	Set ops/s
No simulation	1,796K	1,191K
70ns simulation	1,382K	1,222K
Delta (Get)	-23.0%	+2.6%

Analysis:

The simulated latency reduces concurrent Get throughput by 23% and single-threaded Get by 42%. The observed impact (42.4% p50 increase) exceeds the theoretical prediction: with thread-local routing, approximately 50% of single-threaded Gets access the remote node, so the expected average increase is ~35ns (50% × 70ns) on a ~720ns baseline (~5%). The discrepancy is caused by (a) the `rdtsc` busy-wait calibration in QEMU overestimating cycles-per-nanosecond due to imprecise VM timekeeping, (b) the `_mm_pause()` instructions in the spin loop adding ~5-10ns per iteration beyond the target delay, and (c) the spin loop itself disrupting the CPU pipeline and cache state. **This section validates the simulation mechanism, not the latency magnitude.** The mechanism correctly discriminates between local and remote access. When calibrated on real hardware with stable TSC, the injected delay will match the target value, and the resulting throughput impact will reflect genuine NUMA overhead.

5.9 Limitations

QEMU does not simulate hardware NUMA latency. All memory accesses within the QEMU VM have identical latency regardless of which NUMA node the data resides on. The measured cross-node overheads (4-11.7%) reflect scheduling artifacts, not genuine memory latency differences. On real multi-socket hardware, cross-node access overhead is typically 30-100ns (Intel Xeon: ~70ns, AMD EPYC: ~50ns) on top of ~100ns local access. This means: (a) the p50 NUMA overhead measured in QEMU underestimates the real hardware signal, (b) results like §5.5 where cross-node is *faster* than local are impossible on real hardware, and (c) the cross-VM baseline comparison (§5.7) captures the architectural benefit (lock partitioning) but not the hardware NUMA cost. Real

hardware validation **was load-bearing** for the thesis and has been completed (§5.18, §5.19) — the current results now validate both the methodology/architecture and the performance claims.

Lock-free reads are conditioned on a read-heavy workload pattern (Phase 1). `Get()` calls `unordered_map::find()` without synchronization. This is safe when no concurrent `insert()` targets the same shard, because `find()` on a stable map is thread-safe. The benchmark warmup-then-read pattern guarantees this condition. In production mixed read/write workloads where concurrent `Set()` inserts new keys into the same shard, `find()` may observe a partially-constructed map state. **Phase 2a resolves this limitation**: CMap provides safe concurrent reads during writes via the seqlock protocol (§2.7.5). Phase 1 results in §5.4, §5.6, and §5.7 are valid under the warmup-then-read pattern; Phase 2a results in §5.13–§5.16 are valid under concurrent read/write workloads.

512-byte cross-node overhead remains noisy under QEMU. Requires real hardware validation.

Cross-node reads in Phase 1 execute on the calling thread (synchronous). Phase 2 NodeJob dispatch will enable remote-node reads via pinned workers, potentially reducing cross-node overhead.

Memory fragmentation from the bump allocator is not yet measured.

The per-key `unique_ptr<SeqLock<KeyMeta>>` heap allocation overhead was eliminated in Phase 2a: CMap stores values inline in cacheline-aligned slots with no per-key heap allocation or pointer indirection.

The inline baseline comparison (same VM, same process) shows Furrballs slower than the non-NUMA baseline (0.65x concurrent Set, 0.81x concurrent Get). The cross-VM comparison (§5.7) shows the opposite (3.24x concurrent Set, 1.20x concurrent Get). The discrepancy is likely due to the non-NUMA baseline benefiting from the NUMA VM's 4 vCPUs without paying sharding overhead, while in a dedicated non-NUMA VM (1 node, less vCPU scheduling flexibility), the single-lock baseline bottlenecks. Real hardware validation (§5.18) resolved this for the concurrent Set dimension (1.74x advantage confirmed); the concurrent Get dimension remains affected by the baseline's unsynchronized `unordered_map::find()` (see §5.18 baseline comparison discussion).

5.10 Ablation Study (QEMU VM, 3 runs averaged)

To isolate the individual contribution of each architectural decision, we construct five configurations where each step adds one design change on top of the previous. All configurations use the same benchmark methodology: 64-byte values, SeqLock reads, per-op timing. Single-threaded results use 1 thread (10,000 ops, 5 iterations); multi-threaded results use 2 threads (one per NUMA node, 5,000 keys each, 5 iterations). Each configuration is run 3 times and results are averaged.

Step	Configuration	ST Set p50 (ns)	ST Get p50 (ns)	Self-Get p50 (ns)	Cross-Get p50 (ns)	Cross-OH
A	Baseline: `malloc`, single map, single mutex	811	755	869	870	0.0%
B	+ Topology-aware allocation (per-domain pages, single map)	827	750	854	842	-1.4%
C	+ Per-node sharding (separate maps + mutexes per node)	830	831	1,070	1,050	-1.9%
D	+ Thread-local routing (`GetCurrentNode()` placement)	1,113	1,061	1,193	1,411	**18.3%**
E	+ Shared-nothing cross-node reads (MPSC queue)	1,189	1,134	1,399	2,989	**113.9%**

Step-by-step analysis:

A→B (Topology-aware allocation): Replacing `malloc` with `Numatic::AllocateOnNode()` for

page data produces negligible overhead: ST Set +2% (811→827ns), ST Get -0.7% (755→750ns). This is the expected result in QEMU, which does not simulate hardware NUMA latency—all memory accesses have identical latency regardless of NUMA node. The allocation change is architecturally invisible without genuine cross-node latency differentiation. Cross-OH remains flat (0.0% → -1.4%), confirming that per-node allocation alone does not create a measurable locality signal in this environment.

B→C (Per-node sharding): Set performance is unchanged (827→830ns), confirming that the single-lock bottleneck in B was not a factor at this thread count. Get p50 increases by 10.8% (750→831ns) due to shard iteration overhead: `Get()` must now search per-node shards when the key's location is unknown, adding map lookup indirection. MT Self/Cross latencies increase proportionally (854→1,070 and 842→1,050) from the same shard search. Cross-OH remains flat (-1.4% → -1.9%), confirming sharding alone does not produce a cross-domain signal. The purpose of this step is architectural (enabling per-node isolation for subsequent routing decisions), not a performance optimization at 2 threads. The lock partitioning benefit of sharding becomes significant at higher thread counts (see §5.7: 3x concurrent Set throughput with 4 threads).

C→D (Thread-local routing): Cross-node overhead jumps from -1.9% to **18.3%** — the first measurable cross-domain signal in the ablation. Under round-robin (C), keys are distributed uniformly across nodes, so self-Get and cross-Get have similar latency regardless of the accessing thread's domain. Thread-local routing concentrates keys on the writing thread's domain, creating genuine locality: self-Get finds the key on the local shard (fast path via optimistic local-first lookup), while cross-Get must search the remote shard. ST latency increases (Set +34%, Get +28%) due to `Numatic::GetCurrentNode()` syscall overhead per Set and the less uniform key distribution. This step is where the architecture transitions from topology-agnostic infrastructure to topology-aware behavior.

D→E (Shared-nothing): Cross-Get p50 increases by 112% (1,411→2,989ns), while local operations see a modest increase (Set +7%, Get +7%). The ~1,578ns cross-node penalty is the MPSC slot queue round-trip: claim → fill → submit → worker processes → complete → read response. Local operations bypass the queue entirely; the small overhead comes from CPU competition with spin-polling worker threads (2 benchmark threads + 2 workers on 4 vCPUs). On real hardware with dedicated cores, this local overhead would approach zero.

Summary of isolated contributions:

Design Decision	Contribution	Evidence
Topology-aware allocation	**No signal in QEMU** (+2% Set, within noise)	Requires real hardware NUMA latency to manifest
+ Per-node sharding	**Architectural enabler** (+11% Get from shard search)	Lock partitioning benefit visible at higher thread counts (§5.7)
+ Thread-local routing	**Exposes cross-domain signal** (+20pp cross-OH)	Locality creates measurable local/remote gap

Design Decision	Contribution	Evidence
+ Shared-nothing queue	**Eliminates cross-node reads** (+112% cross-Get)	Queue cost > direct cross-node access for small values

5.11 Variant Comparison: Shared Memory vs Shared-Nothing (QEMU VM, 3 runs averaged)

To evaluate whether message-passing eliminates cross-node synchronization overhead, we implement a **shared-nothing variant** (`SharedNothingCache`) alongside Furrballs' shared-memory design. Both variants share identical infrastructure: per-node physical allocation, bump allocator, SeqLock on `KeyMeta`, thread-local routing, and the same benchmark methodology. The difference is exclusively in the **cross-node read path**.

Shared-memory (Furrballs): The calling thread directly reads from the remote node's `KeyStore` via `SeqLock::Read()` + `memcpy()`. Cross-node cache line access occurs on the calling thread.

Shared-nothing (`SharedNothingCache`): Each NUMA node runs a dedicated worker thread (pinned, spin-polling). Cross-node reads are submitted via a slot-based MPSC queue: the calling thread claims a slot, fills the request, marks it submitted, and spin-waits for the remote worker to process it. The worker does `KeyStore::find()` + `SeqLock::Read()` + `memcpy()` entirely on local memory, then writes the response to the slot. Local reads and writes are identical to Furrballs (direct access, no queue).

Queue design: 256 `alignas(128)` request slots per node. Each slot has a 4-phase lifecycle: `FREE` → `CLAIMED` → `SUBMITTED` → `COMPLETED` → `FREE`. Sender CAS-claims a free slot, fills key data, stores `SUBMITTED` (release). Worker polls for `SUBMITTED` (acquire), processes, stores `COMPLETED` (release). Sender spin-waits on `COMPLETED` (acquire). The release-acquire pattern guarantees visibility of all writes across the slot boundary.

Single-threaded 64B (3 runs averaged):

Variant	Set p50 (ns)	Get p50 (ns)
Furrballs (shared+SeqLock)	~935	~940
SharedNothing (MPSC queue)	~1145	~1120
Delta	+22%	+19%

Multi-threaded 64B, 2 threads (one per node, 3 runs averaged):

Variant	Self-Get p50 (ns)	Cross-Get p50 (ns)	Cross-node Overhead	Optimistic Hit Rate
Furrballs (shared+SeqLock)	~1100	~1100	~0%	0% (round-robin)

Variant	Self-Get p50 (ns)	Cross-Get p50 (ns)	Cross-node Overhead	Optimistic Hit Rate
SharedNothing (MPSC queue)	~1220	~2630	**~110%**	~67%

Analysis:

The shared-nothing variant exhibits three distinct behaviors:

1. **Local-path overhead (+11-22%):** The SharedNothing variant is slower even on local operations. This is a QEMU artifact: the spin-polling worker threads compete for CPU time with the benchmark threads (2 workers + 2 benchmark threads on 4 vCPUs). On real hardware with dedicated cores per NUMA node (typical in server configurations), this overhead would be eliminated—the worker would run on a dedicated core with no benchmark thread competition.

2. **Cross-node queue round-trip (~1,500ns):** The cross-node overhead is the cost of the MPSC slot protocol: claim slot → fill → submit → worker processes → complete → sender reads response. This is a fixed protocol cost independent of data size. The ~1,500ns overhead is dominated by the two `atomic` store/load round-trips and the `yield()` spin-wait. **Design flaw:** The MPSC queue uses CAS-based slot claiming and a 4-phase lifecycle designed for concurrent multi-producer contention. However, the structural single-writer guarantee (exactly one shard holds the key) means there is never concurrent multi-producer access to a single worker's queue for a given key. The multi-producer machinery is pure overhead solving a problem that does not exist in this architecture. A broadcast-race model (§4.4) eliminates the queue entirely.

3. **Optimistic hit rate (67% vs 0%):** The SharedNothing variant's local-first lookup succeeds 67% of the time. This exceeds the naively expected 50% (equal self/cross reads) because the local-first check itself is cheaper than the queue path, and the benchmark interleaves ST warmup writes (all-local) with MT reads. Furrballs with round-robin routing shows 0% because round-robin distributes keys uniformly across nodes—local-first prediction has no correlation with key placement.

Projected real-hardware behavior:

QEMU does not simulate cross-node memory latency. On real hardware (Intel Xeon, ~70ns per cross-node cache miss), the tradeoff shifts:

Cost Component	Furrballs (shared)	SharedNothing (queue)
Local read	~650ns (baseline)	~650ns (identical)
Cross-node read	~650ns + 3-5 × 70ns = ~860-1000ns	~650ns + ~1,500ns queue round-trip = ~2,150ns

Cost Component	Furrballs (shared)	SharedNothing (queue)
Local write	~650ns + mutex overhead	~650ns + mutex overhead (identical)

For small values (64B), Furrballs' direct cross-node access (~860-1000ns) is cheaper than the queue round-trip (~2150ns). The shared-nothing variant becomes competitive when the number of cross-node cache line misses per operation exceeds ~21 (the break-even point where cumulative cache miss latency exceeds the queue round-trip). This threshold is reached for: - Large values spanning many cache lines (4KB+ values) - Complex nested data structures requiring multiple pointer dereferences - SeqLock retry storms under high write contention (each retry incurs additional cache misses)

The analytical tradeoff:

$$\text{SharedNothing wins when: } n_{\text{misses}} \times \text{NUMA\_latency} > \text{queue\_roundtrip}$$

With typical Xeon values: $n_{\text{misses}} > 1500\text{ns}/70\text{ns} \approx 21$ cache misses per operation.

Documented limitation: The QEMU evaluation captures the queue round-trip floor (~1,500ns, a protocol cost) and the local-path overhead (likely inflated by CPU competition). It does not capture the cross-node cache miss cost that the queue is designed to avoid. The analytical projection above estimates real-hardware behavior; validation requires real multi-socket hardware.

5.12 Tradeoff Model

NUMA awareness in Furrballs involves the following measurable tradeoffs:

Dimension	Cost	Benefit
Single-threaded latency	Set +3%, Get +17% overhead (shard iteration cost + per-node indirection)	Data placed on requesting thread's NUMA node
Concurrent read throughput	Modest (1.2x advantage from lock partitioning, offset on real hardware by NUMA latency)	Per-node shards reduce map size, improve cache locality
Concurrent write throughput	None measured (3x advantage from lock partitioning)	N separate mutexes vs 1, reducing contention proportionally
Memory overhead	One physical block per NUMA node, pre-allocated	No per-page syscall overhead, deterministic memory layout

Dimension	Cost	Benefit
Synchronization freedom	`unordered_map::find()` not safe during concurrent `insert()` (Phase 1); **resolved in Phase 2a** via CMap seqlock protocol (§2.7.5)	Lock-free reads with deterministic consistency via seqlock
Shared-nothing variant (local)	+11-22% overhead in QEMU (CPU competition from spin-polling workers; expected ~0% on real hardware with dedicated cores)	Worker exclusively owns node state; no synchronization needed for local ops
Shared-nothing variant (cross-node)	~1,500ns queue round-trip per cross-node read	Eliminates all cross-node cache line misses; wins when misses × 70ns > 1,500ns

The net tradeoff is favorable when: (a) concurrent writes dominate (lock partitioning benefit), or (b) the working set exhibits locality (most accesses hit the local node). It is unfavorable for: (a) uniform random access across all nodes (every access is remote), or (b) single-threaded latency-critical Get workloads where 17% shard iteration overhead is unacceptable. The shared-nothing variant is favorable for workloads with high cross-node miss rates (>21 cache misses per operation on Xeon) and unfavorable for small-value cross-node reads where the queue round-trip exceeds the cumulative NUMA latency.

5.13 Phase 2a: Single-Threaded Performance (QEMU VM, 5 iterations averaged)

Phase 2a replaces the Phase 1 `unordered_map` + `SeqLock` + `shared_mutex` stack with CMap + ConcurrentARC (§2.7, §2.8). This section measures the architectural overhead of the concurrent Swiss table and ARC policy against Phase 1's simpler data structures.

Value Size	Operation	Phase 1 p50 (ns)	Phase 2a p50 (ns)	Delta
64B	Set	695	981	+41%
64B	Get	655	862	+32%
512B	Set	760	895	+18%
512B	Get	672	785	+17%
4KB	Set	1,561	1,649	+6%
4KB	Get	799	946	+18%

Analysis: Phase 2a is consistently 6–41% slower than Phase 1 in single-threaded latency. The overhead comes from three sources: (1) Swiss table probing (SIMD group scan + fingerprint comparison) vs `unordered_map`'s direct bucket lookup, (2) seqlock protocol (CAS even→odd →

write → odd→even) vs Phase 1's SeqLock read, and (3) ARC list management on every Set (SpinLock acquire/release, list splice operations). The 4KB Set delta (+6%) is the smallest because the data copy dominates the fixed overhead of the Swiss table probe. On real hardware with genuine NUMA latency, the ARC eviction benefit (evicting wrong-placed data to make room for well-placed data) should outweigh this overhead for sustained workloads.

5.14 Phase 2a: Concurrent Throughput (QEMU VM, 4 threads, 5 iterations averaged)

Value Size	Mode	Phase 2a Set (ops/s)	Phase 2a Get (ops/s)	Phase 1 Set (ops/s)	Phase 1 Get (ops/s)
64B	All local	1,203,019	1,216,877	761,315	1,197,488
64B	All cross-node	1,429,497	1,441,187	952,933	1,592,695

Analysis: Concurrent Set throughput improved by 50–58% over Phase 1. The mechanism is CMap's per-slot CAS: concurrent writes to different keys proceed in parallel without shared_mutex serialization. Phase 1's `shared_mutex` serialized all writes to the same node shard; Phase 2a's CMap allows concurrent writes to different slots within the same CMap, limited only by the SpinLock for ARC list mutations.

Concurrent Get throughput is within 2% of Phase 1 for local access but 10% lower for cross-node. The PromoteBuf's `fetch_add` on a shared writeHead creates cache line bouncing that Phase 1's lock-free SeqLock reads did not. The cooperative drain (every 64th call) adds occasional `try_lock` overhead. The net effect is small for local access (amortized across 64 operations) but amplified by QEMU's scheduling for cross-node access.

PromoteBuf recovery: Before PromoteBuf was added, ConcurrentARC's Find() took a SpinLock for every ARC promotion, reducing concurrent Get to 0.56x vs the inline baseline (987K ops/s vs baseline's 1,752K). Introducing PromoteBuf with string-based promotions recovered this to 0.73x (1,331K ops/s). Switching to hash-keyed ArcList (eliminating string copies in the promotion path) yielded the final 0.80x (1,217K ops/s) reported above. The 0.56x → 0.80x recovery confirms that the SpinLock on the read path was the dominant bottleneck, and that deferred promotions via a lock-free MPSC buffer effectively eliminate it.

5.15 Phase 2a: Baseline Comparison (QEMU VM, 5 iterations averaged)

Phase 2a comparison against the same non-NUMA BaselineCache from §5.7, using both inline (same VM, same process) and cross-VM methodology.

Inline single-threaded 64B:

Implementation	Set p50 (ns)	Get p50 (ns)
Furrballs (Phase 2a)	981	862
Baseline (non-NUMA, inline)	1,408	1,082
Delta	-30.4%	-20.3%

Inline concurrent 4-thread 64B:

Implementation	Set ops/s	Get ops/s
Furrballs (Phase 2a)	1,203K	1,217K
Baseline (non-NUMA, inline)	1,677K	1,525K
Ratio	0.72x	0.80x

Cross-VM concurrent 4-thread 64B (3 runs averaged):

Implementation	Set ops/s	Get ops/s
Furrballs (Phase 2a, NUMA VM)	916K	1,129K
Baseline (non-NUMA VM)	353K	1,582K
Ratio	**2.60x**	0.71x

Analysis: The inline baseline now shows Furrballs 20–30% faster than baseline in single-threaded (vs Phase 1's +3%/+17%). This inversion is because CMap's Swiss table probe is more efficient than `unordered_map`'s bucket chain for single operations. However, the concurrent inline baseline still outperforms Furrballs (0.72x Set, 0.80x Get) because the non-NUMA baseline benefits from the NUMA VM's 4 vCPUs without per-node sharding overhead.

The cross-VM comparison maintains the Phase 1 pattern: Set advantage (2.60x) from lock partitioning, Get disadvantage (0.71x, worse than Phase 1's 1.20x) from CMap probe overhead. The Get regression is specific to QEMU's uniform memory latency — on real hardware, Furrballs' per-node data locality should partially compensate.

Inline vs cross-VM discrepancy: The inline comparison shows Furrballs 20–30% faster in single-threaded and 0.72–0.80x in concurrent; the cross-VM comparison shows 2.60x Set advantage and 0.71x Get disadvantage. These tell opposite stories for Get performance. The inline baseline runs in the same NUMA VM process, benefiting from the VM's 4 vCPUs without paying sharding overhead. The cross-VM baseline runs in a dedicated non-NUMA VM with less vCPU scheduling flexibility, where the single-lock baseline bottlenecks.

Cross-VM Set regression vs same-VM improvement: Phase 2a's concurrent Set improved +58% over Phase 1 in the same-VM comparison (§5.14: 1,203K vs 761K), but regressed -26% in the cross-VM comparison (916K vs Phase 1's 1,240K from §5.7). The same-VM comparison measures Furrballs Phase 2a vs Phase 1 under identical conditions; the cross-VM comparison measures against

a different baseline in a different VM. The cross-VM baseline's Set throughput dropped from 383K (Phase 1 era) to 353K (Phase 2a era), but Furrballs' absolute throughput also dropped (1,240K → 916K), likely due to CMap's fixed overhead (seqlock protocol, Swiss table probe) being amplified by QEMU's emulation overhead on each CAS instruction. On real hardware, the per-CAS overhead would be ~10ns rather than ~600ns, and the +58% same-VM improvement should dominate.

5.16 Phase 2a: Routing Strategy Comparison (QEMU VM, 2 threads, 5 iterations averaged)

Phase 2a (CMap + ConcurrentARC):

Metric	Round-robin (ns)	Thread-local (ns)	Improvement
Set p50	1,476	1,101	**25.4%**
Set p99	5,057	3,343	**33.9%**
Self-Get p50	891	862	**3.3%**
Self-Get p99	1,756	1,498	**14.7%**
Cross-Get p50	869	929	-6.9%
Cross-Get p99	1,718	2,026	-17.9%

Analysis: Thread-local routing improvement is significantly smaller than Phase 1's 26–41%. The `thread_local` cache in `GetCurrentNode()` (§3.15) eliminated the ~1000ns VM exit per call, making both routing strategies fast. The remaining 3.3% self-Get improvement comes from the local-first probe optimization in CMap (fewer SIMD group scans when probing the local shard first). The cross-Get regression (-6.9%) is a structural consequence of thread-local key placement: with round-robin, ~50% of cross-reads find the key on the local shard (it was round-robined there), avoiding a remote probe entirely. With thread-local routing, cross-reads always miss the local shard and must scan the remote shard — a guaranteed two-shard path versus round-robin's 50% one-shard / 50% two-shard mix.

5.17 Phase 2a: Ablation Study (QEMU VM, 3 runs averaged)

The same five-step ablation structure as §5.10, re-measured with Phase 2a's CMap + ConcurrentARC stack.

Step	Configuration	ST Set p50 (ns)	ST Get p50 (ns)	Self-Get p50 (ns)	Cross-Get p50 (ns)	Cross-OH
A	Baseline: `malloc`, single CMap, single SpinLock	817	824	850	844	-0.8%

Step	Configuration	ST Set p50 (ns)	ST Get p50 (ns)	Self-Get p50 (ns)	Cross-Get p50 (ns)	Cross-OH
B	+ Topology-aware allocation (per-domain pages, single CMap)	934	795	977	915	-6.3%
C	+ Per-node sharding (separate CMaps per node)	984	938	1,032	1,007	-2.5%
D	+ Thread-local routing (`GetCurrentNode()` with `thread_local` cache)	924	859	1,049	1,020	-2.8%
E	+ Shared-nothing cross-node reads (MPSC queue)	1,020	919	934	2,354	+152.1%

Step-by-step analysis:

A→B (Topology-aware allocation): ST Set increases by 14% (817→934ns) — larger than Phase 1's +2% — because CMap's allocator interface calls `Numatic::AllocateLocal` for both ctrl and slot arrays, adding NUMA allocation overhead on top of the baseline's `malloc`. ST Get actually *improves* by 3.5% (824→795ns), likely due to QEMU scheduling artifacts.

B→C (Per-node sharding): ST Set +5% (934→984ns), ST Get +18% (795→938ns). The sharding overhead is larger than Phase 1's +11% Get because CMap's Swiss table probe is more

expensive per-shard than `unordered_map::find()`, and the shard iteration now probes N separate CMaps.

C→D (Thread-local routing): Unlike Phase 1's +34% Set increase from `GetCurrentNode()` syscall overhead, Step D shows a *decrease* in both Set (-6%, 984→924ns) and Get (-8%, 938→859ns). The `thread_local` cache (§3.15) eliminated the syscall, and thread-local routing provides better cache locality for the Swiss table's ctrl array when keys are concentrated on one node. Cross-OH remains flat (-2.5% → -2.8%), confirming the cross-domain signal is still invisible in QEMU.

D→E (Shared-nothing): Cross-Get p50 increases by 130% (1,020→2,354ns), consistent with Phase 1's +112% from the MPSC queue round-trip. The absolute overhead is lower (1,334ns vs 1,578ns) because CMap's lock-free local read is faster than Phase 1's SeqLock read.

Key differences from Phase 1 ablation:

Difference	Phase 1	Phase 2a	Explanation
B→C Get overhead	+11%	+18%	CMap probe cost > unordered_map::find per shard
C→D Set delta	+34%	-6%	`thread_local` cache eliminates syscall
C→D Get delta	+28%	-8%	Same as above
Cross-OH at step D	+18.3%	-2.8%	Thread-local cache makes routing cheap; cross-domain signal remains invisible

5.18 Real Hardware Validation (c6i.metal, AWS, 10 iterations averaged)

All previous evaluation (§5.2–§5.17) was conducted on a QEMU VM with 2 NUMA nodes and 4 vCPUs. QEMU does not simulate real cross-node memory latency (§5.9); the measured effects reflected scheduling artifacts rather than hardware memory access costs. This section presents results from real multi-socket hardware.

Platform: AWS c6i.metal (spot), Intel Xeon Platinum 8375C @ 2.90GHz, 2 sockets, 32 cores/socket, 128 vCPUs (HT), ~257 GB RAM, NUMA distance matrix 10/20 (2:1 local:remote ratio). Ubuntu 24.04, GCC 14.2.0, Release build, no simulated NUMA latency. 10 iterations averaged per test.

Thread placement: Benchmark threads are pinned to specific NUMA nodes via `PinCurrentThreadToNode()` for controlled measurement. This is strictly a measurement technique — Furrballs handles NUMA internally without requiring callers to pin threads. See §5.1 for the full discussion.

Single-Threaded 64B

Operation	p50 (ns)	p99 (ns)	Throughput (ops/s)
Set	141	2,053	4,571,928
Get	108	1,701	5,442,369

Real hardware is **6.8x faster (Set)** and **7.9x faster (Get)** than QEMU at p50, consistent with QEMU's ~600ns per-operation emulation overhead.

NUMA Effect (2 threads, one per socket)

Value Size	Metric	Local (ns)	Cross-Node (ns)	Overhead
64B	p50	504	527	**+4.5%**
512B	p50	551	589	**+7.0%**

Real hardware confirms the cross-domain signal exists at p50 (4.5–7.0% cross-node overhead). The QEMU measurement of 11.7% (§5.4) was inflated by scheduling artifacts; real hardware shows a smaller but genuine signal. The 64B result is noisier due to OS jitter on the bare-metal instance.

Routing Strategy (2 threads, one per node, 64B)

Strategy	Self-Get p50	Cross-Get p50	Self-Improve	Cross-Improve
Round-robin	553ns	565ns	—	—
Thread-local	195ns	224ns	**+64.7%**	**+60.4%**

Thread-local routing on real hardware produces **60–65% improvement**, far exceeding QEMU's +3.3% (§5.16). The `thread_local` cache for `GetCurrentNode()` eliminates the syscall overhead (~1–10ns via VDSO on bare metal vs ~1000ns VM exit in QEMU). This confirms that Phase 1's routing findings were environment-specific (§5.16), not architectural.

Baseline Comparison

Single-threaded 64B:

Implementation	Set ops/s	Get ops/s	Set p50	Get p50
Furrballs	4,618,266	5,494,321	139ns	107ns
Baseline (no NUMA)	8,637,750	13,590,215	110ns	69ns
Delta	-27.0%	-55.4%		

Concurrent 4-thread 64B:

Implementation	Set ops/s	Set ratio
Furrballs	3,831,728	**1.74x**
Baseline	2,205,136	1.00x

The concurrent Set ratio of **1.74x** is the fair comparison: both implementations take locks on the write path, and Furrballs wins through lock partitioning. At higher thread counts (§5.19), this advantage ranges from 1.40x (64 threads) to 5.28x (8 threads) as the baseline's single mutex collapses under contention.

Concurrent Get scaling is reported separately in §5.19 (thread scaling curves) rather than as a ratio here, because the baseline's concurrent Get uses unsynchronized `unordered_map::find()` — the same data race documented in §5.15 — producing anomalously high numbers (19.8M ops/s) that reflect UB rather than legitimate throughput. Furrballs' Get throughput scales +72% from 4→64 threads (5.9M→10.1M ops/s), validating lock partitioning for reads. The baseline's Get numbers (13.4M→23.3M ops/s) also scale because the data race scales with thread count; these numbers are included in the benchmark data file but not presented as a ratio to avoid implying a legitimate comparison.

Ablation Study (Real Hardware)

Step	ST Set (ns)	ST Get (ns)	MT Self (ns)	MT Cross (ns)	Cross-OH
A: Baseline	134	98	234	237	+1.2%
B: + NUMA alloc	134	96	239	235	-1.7%
C: + Sharding	174	133	639	674	+5.6%
D: + Thread-local	141	128	512	619	**+21.0%**
E: + Shared-nothing	135	111	434	1604	+270.0%

Step D (thread-local routing) produces **+21.0% Cross-OH on real hardware** — the largest single contributor to the cross-domain signal. This confirms the thesis: thread-local routing ensures that self-reads hit the local domain, making the cross-node penalty measurable. The absolute Cross-OH varies between runs (+21.0% vs +38.9% in Run 1) due to EC2 spot instance hardware sharing, but the qualitative pattern (Step D is the largest positive contributor) is stable.

Real hardware vs QEMU (ablation step D): QEMU showed -2.8% Cross-OH at step D (NUMA signal invisible). Real hardware shows +21.0% to +38.9%. This is the definitive validation that the architecture's topology-aware benefit requires real cross-domain memory latency to manifest.

Zipfian Workload (theta=0.99)

Strategy	Self-Get p50	Cross-Get p50	Cross-OH	Optimistic Hit Rate
Round-robin	383ns	382ns	-0.2%	0.0%
Thread-local	209ns	232ns	**+10.8%**	50.0%

Thread-local routing under skewed access produces a 10.8% cross-node overhead, confirming the cross-domain signal persists under realistic workload distributions. The 50% optimistic hit rate means half of all cache hits land on the local node.

5.19 Thread Scaling (c6i.metal, 10 iterations averaged)

Per-node lock partitioning creates N independent contention domains. This section tests the hypothesized scaling advantage: Furrballs should scale more smoothly than a single-lock cache because contention grows as `threads_per_node` rather than `total_threads`.

64-byte values, thread-local routing, local Gets. Fresh FurrBall per thread count. Threads distributed round-robin across 2 NUMA nodes (e.g., 64 threads = 32/node).

Threads	Threads/Node	Set ops/s	Get ops/s	Set Scaling	Get Scaling
4	2	3,875,618	5,891,794	1.00x	1.00x
8	4	3,334,534	7,221,196	0.86x	1.23x
16	8	2,365,749	9,033,473	0.61x	1.53x
32	16	1,323,892	9,227,251	0.34x	1.57x
64	32	810,975	10,127,801	0.21x	1.72x

Get throughput scales positively (+72% from 4→64 threads). CMap's seqlock-based Find is nearly contention-free; more threads = more parallel reads. This validates the lock partitioning thesis for reads: per-node sharding prevents cross-node contention on the read path.

Set throughput degrades (-79% from 4→64 threads). All writes on a node contend on the same SpinLock. At 32 threads/node, SpinLock contention dominates. This is the expected bottleneck for a single SpinLock per node; the documented upgrade paths are striped SpinLocks (multiple locks per CMap, keyed by hash bucket) or per-bucket ARC locks.

Get scaling plateaus at ~16 threads (8/node): 9.0M→9.2M→10.1M beyond 16 threads shows diminishing returns, likely from CMap's shared ctrl array probe traffic and memory bandwidth saturation.

Baseline scaling comparison: A standalone baseline thread-scaling test (single `shared_mutex`, `unordered_map`) confirms the single-lock scaling cliff:

Threads	Furrballs Set	Baseline Set	Advantage
4	3,875,618	803,323	**4.82x**
8	3,334,534	631,245	**5.28x**
16	2,365,749	610,360	**3.88x**
32	1,323,892	592,767	**2.23x**
64	810,975	577,237	**1.40x**

The baseline's Set throughput collapses at 4 threads (803K) and stays flat through 64 threads (~577K). The single mutex serializes all writes regardless of thread count. Furrballs outperforms at every thread count because per-node sharding distributes write contention across N SpinLocks. The advantage shrinks from 4.82x→1.40x as Furrballs' own SpinLocks become contended at higher threads/node ratios, but Furrballs never falls below baseline. This confirms the hypothesized scaling advantage from §7: lock partitioning prevents the single-lock scaling cliff.

Interleave Validation (`numactl --interleave=all`)

To verify that Furrballs' explicit NUMA allocation policy overrides the OS default, the full benchmark was run under `numactl --interleave=all`, which sets the process default memory policy to round-robin across all NUMA nodes.

Metric	Normal	Interleave	Delta
64B Cross-OH (p50)	+4.5%	+9.5%	+5.0pp
512B Cross-OH (p50)	+7.0%	+5.1%	-1.9pp
4t Set ops/s	3,875,618	3,743,241	-3.4%
4t Get ops/s	5,891,794	6,130,803	+4.1%
Ablation D Cross-OH	+21.0%	+21.9%	+0.9pp

Results are nearly identical: Furrballs' `numa_alloc_local()` calls use explicit `mbind()` policies that override the process-level interleave default. This confirms that the library's NUMA placement is self-managed and resilient to external memory policy changes.

Ablation Statistical Stability

The run-to-run Cross-OH variance for ablation step D (21.0% in Run 2 vs 38.9% in Run 1, a 17.9pp gap) reflects EC2 spot instance hardware sharing between runs. Within a single run, per-iteration stability is much tighter:

Step	Cross-OH mean	Cross-OH stddev	min	max
D: Thread-local routing	81.2%	**3.8%**	72.4%	84.9%
C: Per-node sharding	31.8%	19.0%	2.1%	71.4%

Step D's stddev (3.8pp) confirms that the thread-local routing signal is statistically stable within a run. The higher stddev at step C (19.0pp) reflects that without thread-local routing, keys are distributed across nodes by round-robin, and the Cross-OH measurement depends on which shard holds each key — a noisier signal. Step D concentrates keys on the requesting thread's node, producing a consistent and stable cross-node penalty.

5.20 Phase 2b: Multi-Node Policy Comparison (DEFERRED)

> **Status: DEFERRED.** Data collected on c6i.metal (commit `6e734ad`, 10 iterations per cell). Results are valid but their inclusion in the evaluation narrative is delayed because page-level eviction smothers policy differentiation (see Limitation Analysis below). The benchmark will be redesigned to isolate what each mechanism actually contributes.

Raw Data

A1: Cross-Domain Overhead (NUMA Instance) (ARC, thread-local routing, 500 distinct keys, 10 iterations)

Metric	Mean	Range
Local p50	97 ns	96–98 ns
Cross p50	108 ns	106–112 ns
Cross-OH	11.6%	8.2–15.5%

A2: Routing Strategy (ARC, 5000 keys/thread)

Strategy	Self-Get p50	Cross-Get p50	Local-hit rate
Round-robin	652 ns	735 ns	0.0%
Thread-local	509 ns	606 ns	50.0%

A3: Thread Scaling (Set throughput)

Threads	ARC ops/s	REMARC ops/s	AUG-ADAPT ops/s
4	21.9M	15.0M	13.8M
8	30.0M	26.7M	33.4M
16	60.1M	53.9M	61.8M
32	91.5M	79.0M	96.4M
64	161.9M	142.8M	154.4M

Policy Comparison: 10% capacity (16 pages, 960 hot keys)

Workload	HR (all policies)	ARC p50	REMARC p50	AUG p50	REMARC mig
ZIPF (theta=0.99)	19.7%	190 ns	70 ns	72 ns	0
LOOP (period=2000)	100.0%	520 ns	352 ns	313 ns	48
TEMPORAL (shift at 50k)	19.4%	186 ns	72 ns	71 ns	0
SCAN-RES (50% scan)	79.9%	412 ns	100 ns	99 ns	0

Policy Comparison: 3% capacity (4 pages, 192 hot keys)

Workload	HR (all policies)	ARC p50	REMARC p50	AUG p50	REMARC mig
ZIPF	3.6%	194 ns	188 ns	210 ns	0
LOOP	23.1%	191 ns	186 ns	190 ns	48
TEMPORAL	4.0%	87 ns	193 ns	185 ns	0
SCAN-RES	12.0%	179 ns	98 ns	81 ns	0

Latency Breakdown: 10% cap, ZIPF (representative single iteration)

Policy	local-hit p50	remote-hit p50	miss p50	lh count	rh count	mi count
ARC	423 ns	399 ns	188 ns	9921	9740	80339
REMARC	218 ns	6184 ns	60 ns	9921	9740	80339
AUG-ADAPT	208 ns	6749 ns	62 ns	9921	9740	80339

Latency Breakdown: 10% cap, SCAN-RES

Policy	local-hit p50	remote-hit p50	miss p50	lh count	rh count	mi count
ARC	546 ns	529 ns	187 ns	39927	39978	20095
REMARC	85 ns	6026 ns	69 ns	39927	39978	20095
AUG-ADAPT	84 ns	6816 ns	69 ns	39927	39978	20095

Ablation Study (ZIPF, 10% cap, 10 iterations)

Step	Policy	Avg p50	StdDev	Min	Max
A	ARC (baseline)	186 ns	8.3 ns	181 ns	209 ns
B	REMARC scoring only	70 ns	1.6 ns	68 ns	73 ns
C	+ migration	70 ns	2.0 ns	68 ns	74 ns
D	+ self-tuning (AUG)	69 ns	0.9 ns	68 ns	70 ns

- Delta B–A (TempCtrl scoring): **-117 ns**
- Delta C–B (Migration): 0 ns
- Delta D–C (Self-tuning): -1 ns

Latency Gap Investigation (ZIPF, 10% cap, 10 iterations per run)

Run	Condition	ARC p50	REMARC p50	AUG p50
A	NoScan (UpdateMinDesire disabled)	182 ns	70 ns	70 ns
B	ReverseOrder (AUG first, ARC last)	184 ns	71 ns	70 ns
C	EqualScan (ARC gets page-warming)	176 ns	69 ns	69 ns

Gap persists in all three control runs. Verdict: **real structural difference** (ArcList pointer-chasing vs TempCtrl contiguous array).

Single-Node Control (no thread-local routing, 100k ops)

Workload	Capacity	HR (all)	ARC p50	REMARC p50	AUG p50
ZIPF	10%	25.6%	56 ns	60 ns	60 ns
LOOP	10%	100%	84 ns	85 ns	84 ns
TEMPORAL	10%	22.5%	55 ns	56 ns	56 ns
SCAN-RES	10%	68.2%	81 ns	99 ns	98 ns
ZIPF	3%	4.4%	55 ns	56 ns	56 ns
LOOP	3%	23.2%	55 ns	56 ns	56 ns
TEMPORAL	3%	4.1%	55 ns	56 ns	56 ns
SCAN-RES	3%	23.2%	65 ns	58 ns	60 ns

Single-node: no latency gap. All policies within 1–18 ns of each other.

Limitation Analysis

The data above reveals a fundamental problem with using page-level eviction as the evaluation substrate for cache policy comparison:

1. **Hit rates are identical across all policies in every workload.** Page recycling evicts all keys on a page wholesale. No policy can selectively retain high-value keys. The Locality Barrier (~24.5% ceiling at 10% cap, Finding 1) is not just a ceiling — it is a study validity constraint.

2. **Per-key EvictScore is inert for eviction.** REMARC computes a desire score per key, but this score never selects individual victims. It only influences migration (MigrateScore > MinDesire). Migration fired in exactly one workload (LOOP, 48 migrations). In ZIPF, TEMPORAL, and SCAN-RES: zero migrations.

3. **The 2.6x latency gap is a data structure artifact, not a policy advantage.** The ablation shows the entire gap appears at Step B (adding TempCtrl scoring) with zero contribution from migration or self-tuning. Single-node results confirm no gap exists. The gap is caused by ArcList (pointer-chasing doubly-linked list) vs TempCtrl (contiguous array, cache-line-local access) — a structural difference in metadata layout, not in eviction intelligence.

4. **What the study actually measures:**

   - Multi-node routing overhead (A1: 11.6% cross-node penalty)
   - Data structure latency (TempCtrl array faster than ArcList linked list)
   - Migration benefit on perfectly periodic workloads (LOOP only)

   It does **not** measure "which policy makes better eviction decisions" because no policy gets to make that decision.

5. **Remote-hit latency inversion.** In the latency breakdown, REMARC/AUG-ADAPT show remote-hit p50 of 6000–6800 ns vs ARC's 399 ns. This is because REMARC's faster miss path (68 ns vs 188 ns) means the remote-hit path — which requires cross-node communication — appears relatively slower. The absolute latency is likely similar, but the distribution shifts.

Why Deferred

Including this data as-is in the evaluation would support the narrative that REMARC is "2.6x faster" when in fact:

- Hit rates are identical (no policy advantage in the metric that matters most)
- The latency advantage comes from array vs linked-list metadata layout
- Migration is the only REMARC-specific mechanism that actually fires, and only on one synthetic workload
- The per-key scoring that is REMARC's theoretical contribution never influences eviction

A honest paper cannot claim policy superiority from data structure layout differences. The benchmark must be redesigned to either: (a) isolate data structure overhead from policy decisions, or (b) test on a substrate where policy decisions actually matter (per-key eviction, compaction, or a separate PolicyBench)

What Remains Valid

The following findings from this run are independent of the policy comparison issue:

- **A1 cross-node overhead** (11.6%) is a pure architecture measurement
- **A2 routing strategy** confirms thread-local routing advantage
- **A3 thread scaling** shows linear scaling for all policies
- **Single-node parity** confirms no policy overhead in single-node mode
- **Migration activates on LOOP** (48 migrations, 40% latency reduction) is a real mechanism observation

These can be extracted and included in the evaluation without the misleading policy comparison framing.

5.21 Per-Key Eviction: Design Plan

> **Status: Planned.** This section documents the design decisions, benchmark methodology, and implementation priority for per-key eviction — the change that will make REMARC's EvictScore meaningful and enable fair policy comparison.

Root Cause of §5.20 Deferral

Page-level eviction made hit rates identical across all policies because no policy could selectively retain high-value keys. REMARC's EvictScore was computed per-key but never acted on — the only eviction mechanism was wholesale page recycling. The Locality Barrier was not just a ceiling but a study validity constraint.

The fix: add per-key eviction so the policy's scoring actually determines what stays in cache.

Architecture: Two-Level REMARC

The original intent was always a two-level control structure, where per-key (fine granularity) and per-page (coarse granularity) each provide different decision spaces:

```
Level 1: KeyPolicy (per-key REMARC instance)
  Element:   individual key-value entry within a page
  State:     TempCtrl[256] per page (uint8_t, already exists)
  Atoms:     AtomSLocal (recency), AtomSRemote (frequency)
  Projections: EvictScore(key), MigrateScore(key)
  Actions:   evict key, keep key
  Trigger:   capacity pressure → find min EvictScore key → evict → free slot

Level 2: PagePolicy (per-page REMARC instance / super-policy)
  Element:   page (aggregation of N keys)
```

```
State:     derived from Level 1 outputs (mean EvictScore, occupancy, tier)
Projections: EPage(sum, n) - already exists as RemarcComputeEPage
Actions:   migrate page to node, recycle empty page, compact pages, rebalance keys between pa
Trigger:   periodic scan, NUMA rebalance, memory pressure
```

Level 2's inputs come FROM Level 1's outputs. A page's "health" is the aggregate of its keys' scores. EPage already computes this (RemarcComputeEPage). The two levels compose: KeyPolicy decides which keys survive; PagePolicy decides where pages live and when they're recycled.

This is analogous to Memcached's model: slab = allocation unit (our page), item = eviction unit (our key), slab reclamation = lazy/when-empty (our Level 2 recycle). Furrballs adds REMARC scoring at both levels plus topology-aware placement.

Per-Key Eviction Implementation

Approach: tombstone + dead-bytes, no in-page reuse (simplest viable)

1. When capacity pressure: find the key with the lowest EvictScore across all pages
2. `RemoveKeyEntry(idx)` — removes from KeyIndex/TempCtrl/HotNodes (already exists in Page.h:153)
3. Erase from CMap (`KeyStore.EraseByHash(hp)`) — already exists for page recycling
4. Track dead data bytes per page (the freed entry's data region is unreusable but accounted)
5. Page recycling when `ActiveKeys == 0` — already works (line 1217-1228)

Why no free-list initially: The free-list is infrastructure, not policy. It affects ARC and REMARC equally — both use the same mechanism to reclaim freed slots. The policy difference is only in which key gets selected for eviction. Therefore, the simplest viable implementation (tombstone, no in-page reuse) produces a fair comparison. Free-list optimization (SIMD scan, compact offsets) is deferred to a later phase.

Bump allocator preserved: Bump handles the fast path (fresh page, sequential writes). Tombstoned entries accumulate dead space. When a page empties completely (ActiveKeys == 0), Recycle() resets the bump pointer. New entries go to pages with bump space; pages with high dead-byte ratios drain naturally as their coldest keys are evicted.

Future optimization (SIMD free-list): When the free-list is added, each entry is (offset: uint32_t, size: uint32_t) = 8 bytes. AVX2 YMM register holds 4 entries. First-fit scan becomes vectorized. This matches the existing RemarcScanBatch SIMD pattern. Deferred because it does not affect the policy comparison.

Benchmark Methodology: 2×2 Factorial Design

The previous benchmark compared policies at the same (page-level) granularity — no differentiation possible. The new design is a 2×2 factorial:

```
                    |  1-level (per-key only)  |  2-level (per-key + per-page)
--------------------|--------------------------|------------------------------
ARC scoring         |  ARC-1L: ArcList per-key |  ARC-2L: + page migration/tiering
REMARC scoring      |  REMARC-1L: TC per-key   |  REMARC-2L: + page super-policy
```

Tier 1 (row comparison): ARC-1L vs REMARC-1L Same granularity (per-key), different scoring. Isolates "which scoring mechanism produces better hit rates?" This is the core policy comparison — fair because both operate at the same level.

Tier 2 (column comparison): REMARC-1L vs REMARC-2L Same scoring, different architecture. Isolates "does the page-level super-policy add value beyond per-key decisions?" This tests the two-level architecture thesis.

Cross-comparison: ARC-1L vs REMARC-2L "How much does the full REMARC stack improve over baseline ARC?"

REMARC-BLOOM as speed baseline: Page-level only, Bloom filter for membership, no per-key state. Measures the cost of per-key scoring. This is the fastest variant — useful as a throughput ceiling reference.

ARC Data Structure Finding

Phase 2b revealed a 2.6x latency gap between ARC and REMARC that persists across all investigation controls (§5.20). The gap is caused by ArcList (doubly-linked list, pointer-chasing) vs TempCtrl (contiguous array, cache-line-local).

This is not fundamental to the ARC algorithm. ARC requires: - O(1) promote (move to most-recent position) - O(1) evict (remove least-recent) - O(1) lookup (find key in list)

A flat array with a clock hand or generation counter satisfies all three with better cache behavior. This suggests a three-way comparison that isolates data structure from policy:

```
1. ARC-linked   - current implementation (linked list metadata)
2. ARC-flat     - same ARC algorithm, flat array layout
3. REMARC-flat  - different scoring, flat array layout (current)
```

Deltas: - **ARC-flat vs ARC-linked**: pure data structure benefit (measures the cost of pointer-chasing) - **REMARC-flat vs ARC-flat**: pure policy benefit (measures scoring quality, equalized layout) - **REMARC-flat vs ARC-linked**: combined (total REMARC advantage)

This is a publishable finding independent of REMARC: "ARC's linked list implementation is 2-3x slower than a flat array for cache metadata access in NUMA settings, and this is not fundamental to the adaptive replacement algorithm." ARC's strength (p-adaptation between recency and frequency) comes with a hidden cost (linked list pointer-chasing). REMARC's simpler 2-atom composition avoids this cost by design.

Double-edged sword characterization: - ARC: smarter adaptive mechanism → potentially better hit rates, but slower per-operation metadata access - REMARC: simpler scoring → potentially worse hit rates, but faster per-operation metadata access - The total throughput comparison (hit rate × latency) is workload-dependent and untested with per-key eviction

Implementation Priority

1. **Per-key eviction (simplest viable):** tombstone + dead-bytes, no in-page reuse. Fair comparison, minimal code changes. Estimated: Page.h (~20 lines), Furrballs.cpp (~40 lines).
2. **2×2 benchmark:** re-run Phase 2b with per-key eviction. Same workloads, same methodology. Expect hit rate divergence.
3. **3-way comparison (ARC-linked vs ARC-flat vs REMARC-flat):** requires ARC-flat implementation. Isolates data structure from policy.
4. **Optimization (deferred):** SIMD free-list, compact uint32_t offsets, REMARC-BLOOM speed baseline. These are engineering improvements that do not affect the policy comparison.

5.22 ARC-Flat: Data Structure Improvement to ARC

> **Status: Validated.** Three ARC variants compared in PolicyBench (capacity=1000, universe=10000, 100k ops, 3 iterations). All produce identical hit rates. ARC-FLATLL (flat-linked-list) is 1.5–34% faster than ARC (std::list).

Background

§5.20 identified a 2.6x latency gap between ARC (ArcList with `std::list`) and REMARC (TempCtrl contiguous array). This section investigates whether ARC can be improved by replacing `std::list` with a flat-array-based data structure, preserving the exact ARC algorithm.

Three Variants Tested

ARC (baseline): `std::list<uint64_t>` + `std::unordered_map` index. Per-node heap allocation on every `push_front`. O(1) all operations but cache-hostile (pointer chasing, scattered memory).

ARC-FLAT (dense + scan): Dense `std::vector<uint64_t>` for keys and timestamps. Monotonic clock for ordering: MRU = max timestamp, LRU = min timestamp. `splice_front` = update timestamp (O(1)). `pop_back` = scalar linear scan for min-timestamp (O(n)). No heap allocation. Contiguous memory.

ARC-FLATLL (flat-linked-list): `std::vector<Slot>` where each slot contains `{key, prev_index, next_index, active}`. Doubly-linked list using array indices instead of heap pointers. Free-list for slot reuse. O(1) all operations. No heap allocation. Contiguous memory.

Hit Rate Validation

All three variants produce **bit-identical** hit/miss/eviction counts across all four workloads:

Workload	ARC	ARC-FLAT	ARC-FLATLL
Zipfian (theta=0.99)	34,964	34,964	34,964
Scan-Resistant	88,346	88,346	88,346
Temporal Shift	74,726	74,726	74,726
Looping	0	0	0

This confirms that flat-linked-list with array indices is **semantically equivalent** to `std::list` for the ARC algorithm. The ordering is encoded identically (prev/next links), just in contiguous memory instead of scattered heap nodes.

Throughput Results

Workload	ARC	ARC-FLAT	ARC-FLATLL	FLATLL vs ARC
Zipfian	4.95M	920K	**5.03M**	**+1.5%**
Scan-Resistant	8.05M	4.59M	**10.77M**	**+34%**
Temporal Shift	10.4M	1.72M	**13.6M**	**+31%**
Looping	5.07M	747K	**6.75M**	**+33%**

KPI scoreboard (weighted: Hit 0.60, Ops/s 0.25, EvictEff 0.15): ARC-FLATLL ranked **#3** (62.95), ARC ranked **#6** (60.29).

Why ARC-FLAT (Dense+Scan) is Slow

The O(n) scalar scan for min-timestamp makes eviction ~200x slower than linked list tail access. With capacity=1000 and universe=10000, the cache thrashes constantly (65% miss rate on Zipfian → 63,036 evictions per 100k ops). The eviction path is the hot path, not the cold path. The scalar scan dominates: ~1000ns per eviction vs ~5ns for linked list tail. Net: 5–14x slower.

This would be partially mitigated by SIMD min-reduce (16x scan speedup with AVX2) but the fundamental issue is that eviction is frequent and O(n) is expensive at n=1000.

Why ARC-FLATLL (Flat-Linked-List) is Fast

The flat-linked-list has the same O(1) complexity as `std::list` for all operations, but eliminates heap allocation:

- `push_front`: allocate from free-list (array index) vs `malloc` → ~5ns vs ~50–100ns
- `pop_back`: follow `tail_` index vs `free()` → ~5ns vs ~50–100ns

- `splice_front`: relink via array indices vs pointer manipulation → ~5ns vs ~10ns

The improvement is largest on high-hit-rate workloads (Scan-Res +34%, Temporal +31%) because these workloads have the most `push_front`/`pop_back` operations (frequent ghost list management). When hit rate is high, ARC's ghost lists (B1/B2) are actively used for p-adaptation, triggering many allocation-heavy list operations.

On Zipfian (+1.5%), the improvement is smallest because the cache thrashes so badly (65% miss rate) that most operations are evictions from T1/T2 to B1/B2, not ghost-list-intensive p-adaptation.

Implementation Details

```
struct Slot {
    uint64_t key = 0;
    size_t   prev = SIZE_MAX;   // array index of previous entry
    size_t   next = SIZE_MAX;   // array index of next entry
    bool     active = false;
};

std::vector<Slot> slots_;                          // contiguous array
std::unordered_map<uint64_t, size_t> idx_;         // key → slot index
size_t head_ = SIZE_MAX;                            // MRU entry index
size_t tail_ = SIZE_MAX;                            // LRU entry index
size_t freeHead_ = SIZE_MAX;                        // free-list head index
```

Memory layout: each Slot is ~32 bytes (8 + 8 + 8 + 1 + 7 padding). For capacity=1000: up to 4 lists × 1000 entries = 4000 slots × 32 bytes = 128KB. For Furrballs per-page cache (256 entries): 4 × 256 × 32 = 32KB → fits in L2 cache. Individual list (256 × 32 = 8KB) fits in L1.

Free-list: inactive slots chain through their `next` field. `allocSlot()` pops from free-head; `freeSlot()` pushes to free-head. No malloc/free after initial allocation.

Further Optimization Opportunities

1. **Replace `unordered_map` with open-addressing flat hash** (like CMap). Eliminates per-bucket heap allocation in the hash index. Currently both ARC and ARC-FLATLL use the same `unordered_map`, so the comparison is fair, but a flat hash would improve both.
2. **Pre-allocate fixed capacity**: Reserve exactly 4 × capacity slots upfront (max across T1+T2+B1+B2). No vector growth. Arena-style allocation.
3. **Compact indices to `uint32_t`**: For capacity 4B, `uint32_t` suffices for prev/next. Reduces Slot from 32 to 20 bytes. 60% denser packing → better cache line utilization.

4. **SIMD batch operations**: Not needed for single key operations (already O(1)), but useful for batch scanning (e.g., find all keys with given property, count list sizes).
5. **Remove redundant `unordered_set<uint64_t> cached_`**: Currently BareARC tracks cached membership separately. T1 and T2 together ARE the cache — `cached_` is redundant. Eliminating it saves one hash lookup per access.

Significance for ARC Literature

Every major ARC implementation (ZFS, PostgreSQL, FlashCache, Redis modules) uses linked lists for T1/T2/B1/B2 management, with per-node heap allocation. The flat-linked-list demonstrates that:

1. ARC does not require heap allocation per list operation.
2. A flat array with index-based linking preserves the exact algorithm.
3. Throughput improves 1.5–34% depending on workload hit rate.
4. The improvement is largest where ARC is most valuable (high hit rate, active p-adaptation).

The flat-linked-list is a drop-in replacement for `std::list` in any ARC implementation. It changes ARC's throughput narrative: ARC's adaptivity no longer comes at the cost of heap allocation overhead.

ARC-FLATLL2: Optimized Variant

ARC-FLATLL already eliminated heap allocation and matched ARC's hit rates exactly. The remaining bottlenecks in ARC-FLATLL were: (a) bloated Slot struct (32 bytes due to `size_t` indices), (b) dynamic `vector` growth, and (c) redundant cache membership tracking. Three optimizations were applied:

1. **`uint32_t` indices** instead of `size_t`: Slot shrinks from 32 to 20 bytes (8 key + 4 prev + 4 next + 1 active + 3 padding). 60% denser packing → better cache line utilization. For capacity 4B entries, `uint32_t` suffices. This was the dominant optimization: reducing Slot from 32→20 bytes means 3 slots per 64-byte cache line instead of 2, a 1.5x improvement in cache line efficiency for list traversal operations (splice, erase).
2. **Pre-allocated arena**: Each list pre-allocates 2×capacity slots at construction. Free-list chains inactive slots. Zero malloc/free after construction. Note: the 2× factor is necessary because ARC does not bound individual lists to cap — only the total across all four lists is bounded to 2×cap. A single list (e.g., T2 after sustained ghost-list pressure) can hold up to 2×cap entries. This was discovered via a crash during verification when lists were pre-allocated at exactly cap.
3. **Removed redundant `cached_` set**: The original BareARC implementation tracks cache membership in a separate `unordered_set<uint64_t>`. Since T1 T2 IS the cache, membership is `t1_.contains(k) || t2_.contains(k)` — one redundant hash lookup eliminated per access.

Failed attempt: flat open-addressing hash. We also tried replacing `unordered_map` with a hand-rolled open-addressing hash table (linear probing, tombstone deletion). This made throughput **9x worse** (615K vs 5.6M ops/s on Zipfian). The cause: ARC under thrashing workloads (65% miss rate, 63K evictions per 100K ops) creates extreme churn — constant insert/erase cycling. Tombstones accumulate with no rehash path, degrading probe chains to near-linear. `unordered_map` with `reserve()` avoids this because its bucket-based design handles churn without tombstone pollution. The hash table was never the bottleneck; the Slot struct density was.

Hit rates remain identical (verified across all 4 workloads). Throughput results:

Workload	ARC	ARC-FLATLL	ARC-FLATLL2	vs ARC	vs FLATLL
Zipfian	5.05M	5.60M	**9.45M**	+87%	+69%
Scan-Res	9.53M	11.2M	**19.8M**	+108%	+77%
Temporal	11.2M	12.7M	**21.5M**	+92%	+69%
Looping	5.22M	7.10M	**14.2M**	+172%	+100%

KPI scoreboard: ARC-FLATLL2 ranked **#2** (71.60), behind only AUG-ADAPT (72.87, which benefits from 98% looping hit rate via ARC-specific detection). ARC-FLATLL ranked #4 (61.73), ARC ranked #7 (59.78).

The uint32_t compaction is responsible for the bulk of the improvement (denser cache lines), with pre-allocation and `cached_` removal contributing incrementally. The flat hash experiment confirms that `unordered_map` is not a bottleneck under ARC's churn pattern.

Known Limits and Edge Cases

The following limits were identified and verified via the `--limits` test suite (15/15 pass):

1. **Maximum capacity**: `uint32_t` indices with `NIL = UINT32_MAX` sentinel. Each list pre-allocates 2×cap slots. Valid indices: 0..2×cap-1. Maximum cap before `NIL` collision: 2^31 - 1 2.1B entries. In practice, memory is the binding constraint (4 lists × 2×cap × 20 bytes = 160×cap bytes; cap=1M → 160MB).
2. **Per-list occupancy**: ARC's invariant is |T1|+|T2|+|B1|+|B2| 2×cap. Any single list can theoretically hold 2×cap entries. The pre-allocation of 2×cap per list is necessary (discovered via T2 overflow crash at cap=1000). Verified with ghost-hit pressure test that drives T2 to maximum occupancy.
3. **Minimum capacity (cap=1)**: Works correctly. All list operations degrade to single-element push/pop/erase. Hit rates match ARC exactly (verified with alternating 3-key workload).
4. **cap universe**: No eviction after warmup. All keys fit in cache. Second-pass hits = 100%. Verified.

5. **Pure sequential scan**: Every access misses (capacity « universe). Ghost lists (B1/B2) never fire because no key is revisited. p stays at 0. Hit rates = 0 for all policies. Verified with 5000-key scan at cap=100.
6. **Single key repeated**: After first miss, all subsequent accesses hit. No eviction. cap=1 works correctly (999 hits, 1 miss for 1000 ops). Verified.
7. **High churn workloads**: ARC under thrashing (miss rate > 50%) creates extreme insert/erase cycling on ghost lists. Tombstone-based open-addressing hash fails here (9x slower). `unordered_map` with `reserve()` handles this correctly.
8. **Thread safety**: None. `ArcFlatLinkedList2` and `BareARC_FlatLL2` are single-threaded. Concurrent access requires external synchronization (production Furrballs wraps in per-node SpinLock).

§5.23 ARC-REMARC Hybrid: Negative Result

> **Status: Negative finding.** REMARC per-key TempCtrl scoring does not subsume ARC's p-adaptation. The mechanisms are complementary, not substitutable.

Design: The hybrid preserves ARC's full p-adaptation machinery (T1/T2/B1/B2 ghost lists, p-adjustment on ghost hits) but replaces one operation: eviction victim selection. Instead of always evicting the LRU tail of the chosen list, the hybrid scans the list for the entry with minimum TempCtrl and evicts that one. This tests whether per-key desire scoring makes better eviction decisions than position-based LRU.

REMARC's contribution to the hybrid is minimal — a single `uint8_t tempCtrl` per slot with a simple scoring rule: - On T1 hit (promoted to T2): tempCtrl = 128 (high desire, recently used) - On T2 hit: tempCtrl += 1 (increment, capped at 255) - On cold miss insertion to T1: tempCtrl = 64 (moderate desire, untested) - On ghost hit insertion to T2: tempCtrl = 128 (high desire, ghost-validated) - On eviction: `popMinDesire()` walks the linked list, evicts the slot with lowest tempCtrl

ARC's contribution is unchanged — p-adaptation via ghost lists, list management, the full ARC algorithm. The only change is *which entry within a list* gets evicted.

Results:

Workload	ARC (LRU)	HYBRID (min-Desire)	Delta
Zipfian	34,964	33,151	**-5.2%**
Scan-Resistant	88,346	90,117	**+2.0%**
Temporal Shift	74,726	58,767	**-21.4%**
Looping	0	96,999	**+97%**

KPI: HYBRID ranked #3 (70.44) vs ARC-FLATLL2 #4 (69.77) — essentially tied. The hybrid gains on Scan-Res and Looping but loses catastrophically on Temporal Shift.

Analysis:

1. **Temporal Shift (-21.4%):** The hybrid's Achilles heel. When the workload shifts from one access pattern to another, ARC's ghost lists immediately adjust p (ghost hit → p changes), redirecting future evictions. The hybrid evicts based on TempCtrl, which reflects *cumulative* access frequency, not *current* access pattern. Keys that were hot in phase 1 have high TempCtrl and survive into phase 2, taking cache slots from keys that are hot in phase 2. ARC's LRU eviction handles this naturally — the tail of the list is always the least recently used regardless of past frequency.

2. **Zipfian (-5.2%):** Under thrashing (65% miss rate), the hybrid makes different eviction choices than LRU. Some of these are better (keeping genuinely hot keys), some worse (keeping keys that were hot recently but are now cooling). The net effect is slightly negative. ARC's LRU is robust because it doesn't depend on scoring accuracy — it always evicts the least recent, which is a good default under uncertainty.

3. **Scan-Resistant (+2.0%):** The hybrid correctly protects the hot set. TempCtrl scores for repeatedly-accessed keys grow high (127+), making them nearly invulnerable to eviction. ARC's LRU occasionally evicts a hot key that happened to be at the tail due to timing. The +2% improvement is real but small.

4. **Looping (+97%):** The hybrid achieves 97% hit rate (matching AUG-HEAP) because TempCtrl monotonically increases for the active loop keys. Once the loop keys have tempCtrl 200+, they're never evicted. ARC scores 0% because it has no loop detection — every loop iteration evicts the previous loop's entries. This is a known ARC weakness.

Conclusion: REMARC's per-key scoring and ARC's p-adaptation solve different problems. TempCtrl scoring captures *how desirable* a key is (access frequency). ARC's p-adaptation captures *what balance of recency vs frequency* the workload currently needs. The hybrid only replaces the former, losing the temporal adaptivity that makes ARC robust under phase shifts. The two mechanisms are complementary — a true hybrid would need both operating simultaneously, not one replacing the other.

This finding supports REMARC's positioning as a *multi-action* framework (eviction + migration) rather than a *better eviction* policy. REMARC's value is not in replacing ARC's eviction heuristic but in adding the migration axis that ARC cannot express.

Two follow-up experiments confirmed and refined this conclusion:

- **HYBRID-POS (position-weighted scoring):** Combining position in the linked list with TempCtrl (score = TempCtrl - position_weight) preserves recency, yielding nearly identical hit rates to ARC on Zipfian (+0.2%) and Temporal Shift (+0.02%). But it loses the Looping benefit entirely (5% vs 97%) because position weighting penalizes entries not at the MRU end. KPI ranked #19 (51.24) due to O(n) scoring overhead on every eviction.

- **HYBRID-FB (feedback control):** A reversal-rate feedback controller (switch from min-Desire to LRU when >15% of evicted keys are re-accessed) produced identical results to the base HYBRID. The reversal rate never exceeded the threshold because min-Desire eviction makes *different* choices from LRU, not *worse* choices. The failure mode is opportunity cost, not reversals.

Full analysis and methodology are documented in the REMARC companion paper (Findings 32–34, DOI: 10.5281/zenodo.19794758).

5.24 Corrected-Capacity NUMA Comparison at 1024B (c6i.metal)

> **Status: v1.30.0 — Corrected capacity and fixed ReadOnly prefix bug.** All systems at 32MB usable. Previous runs had unequal capacities (FurrBall 32MB, CacheLib ~16MB usable, CacheLibNuma 64MB due to floor-bump bug). Capacity alignment and footprint tracking added in commit `d6c66c4`. Remote-Only adapter added in `bfc937b`. **v1.29.0 critical bug:** ReadOnly pre-populate used `"key_"` prefix, workload used `"p_"` — all Gets probed empty CMap (43ns miss path, not data reads). Fixed in `f023f52`. **v1.30.0 new results:** corrected ReadOnly data, UniformRO workload type added, working-set sweep (64MB/128MB) completed. NUMA DRAM penalty measured: **171ns at 64MB, 211ns at 128MB**. CacheLib updated to v2026.02.23.00 (identical performance to v2024.07). CacheLib's Memory Tiers feature confirmed **not implemented** in any open-source release.

Platform: AWS c6i.metal (spot), Intel Xeon Platinum 8375C @ 3.5GHz, 2 sockets, 64 cores/socket, 128 vCPUs (HT), ~257 GB RAM, NUMA distance matrix 10/20. Ubuntu 24.04, GCC 14.2.0, Release build. 10 repetitions per benchmark.

Methodology

Capacity alignment. All systems target 32MB usable capacity. FurrBall allocates `pagesPerNode * 4096 * nodeCount` bytes via `numa_alloc_local()` per node (1:1 usable-to-footprint ratio). CacheLib's slab allocator reserves ~50% of configured size for internal metadata (slab headers, free lists, background worker buffers), so we configure CacheLib at 64MB (`setCacheSize(64MB)`) and allocate a single pool of 32MB (`addPool("default", 32MB)`). CacheLibNuma uses 64MB config split into two pools of 16MB each (`addPool("node0", 16MB)`, `addPool("node1", 16MB)`). The `footprint_mb` counter in each benchmark reports actual configured memory.

System	Config Size	Usable Pool(s)	Footprint	Eviction
FurrBall TL (2 nodes)	32 MB (2 x 16 MB pages)	32 MB	**32 MB**	ARC (FlatList)
FurrBall SN (1 node)	32 MB (1 x 32 MB pages)	32 MB	**32 MB**	ARC (FlatList)

System	Config Size	Usable Pool(s)	Footprint	Eviction
FurrBall CN (2 nodes, RoundRobin)	32 MB (2 x 16 MB pages)	32 MB	**32 MB**	ARC (FlatList)
CacheLib	64 MB (slab config)	32 MB (1 pool)	**64 MB**	LRU
CacheLibNuma	64 MB (slab config)	32 MB (2 x 16 MB pools)	**64 MB**	LRU
TBB	unbounded	unbounded	**0** (default new)	None

CacheLib pays a **2x memory tax** for its slab allocator metadata. FurrBall achieves the same usable capacity at half the memory footprint.

Thread placement. Each worker thread is pinned to a NUMA node via `Numatic::PinCurrentThreadToNode(n)`. With 2 threads, thread 0 on node 0, thread 1 on node 1.

Workloads. Each thread processes 100,000 operations. Value size is 1024 bytes (chosen to make remote memory access cost visible — a 1KB memcpy across NUMA UPI costs ~40-60ns, which is detectable vs the ~5ns for 64B).

- *Partitioned (Zipfian, theta=0.99, universe=700K):* Each thread accesses a disjoint key range. Measures per-thread-local latency under NUMA-aware vs NUMA-blind routing.
- *Shared (Zipfian, theta=0.99, universe=700K):* Both threads access the same 700K-key space. Measures contention + cross-node access cost.
- *ReadOnly (pre-populated, Zipfian, theta=0.99, universe=700K):* Keys are inserted before timing starts. During the timed loop, all operations are Gets. **No eviction occurs.** This isolates the pure access path from the eviction path, eliminating the p99_set storms that dominate FurrBall's throughput.
- *UniformRO (pre-populated, uniform random, universe=700K):* Same as ReadOnly but with uniform key distribution instead of Zipfian. Zipfian concentrates on ~30K hot keys (theta=0.99 with 700K universe), which always fit in L3 regardless of cache size. Uniform access spreads reads across all allocated keys, forcing DRAM access when capacity exceeds L3 (56MB on Ice Lake). This is the workload used for the working-set sweep.

Systems tested.

Adapter	Routing	Nodes	Notes
FurrBall TL	ThreadLocal (key hash -> home node)	2	Each thread accesses its own node's CMap
FurrBall SN	Single node (all on node 0)	1	Lock contention baseline
FurrBall CN	RoundRobin (keys split across 2 nodes)	2	~50% remote access per thread

Adapter	Routing	Nodes	Notes
FurrBall LRUTL	ThreadLocal, LRU policy	2	LRU vs ARC comparison
FurrBall LRUSN	Single node, LRU policy	1	LRU lock contention baseline
FurrBall LRUCN	RoundRobin, LRU policy	2	LRU cross-node
CacheLib	Default (single pool, no NUMA binding)	N/A	Accidental NUMA interleaving
CacheLibNuma	Per-pool routing (pool 0 for thread 0, pool 1 for thread 1)	2	Explicit NUMA awareness

NUMA signal isolation. The key comparisons for isolating the NUMA memory penalty:

- **TL vs CN (same system, same locks, different key placement):** TL routes each thread to its local node. CN distributes keys across both nodes. The latency difference is the NUMA penalty.
- **TL vs SN (lock distribution vs lock sharing):** Both use local memory. TL has 2 SpinLocks (one per node). SN has 1 SpinLock. The latency difference is lock distribution benefit.
- **ReadOnly (no eviction):** Eliminates p99_set storms (35-65us in FurrBall). Measures pure CMap probe + memcpy + PromoteBuf latency.

Results: Partitioned 2t, 1024B, 32MB

System	p50 Get	p50 Set	p99 Set	ops/s	Hit%	Footprint	node0	node1
FurrBall LRUTL	79 ns	471 ns	35,846 ns	2.74M	93.9%	32 MB	80	79
FurrBall TL	77 ns	557 ns	36,003 ns	2.61M	93.9%	32 MB	78	76
FurrBall LRUSN	172 ns	1,942 ns	95,117 ns	0.89M	93.9%	32 MB	174	-
FurrBall SN	156 ns	2,012 ns	90,128 ns	0.90M	93.9%	32 MB	156	-
CacheLib	198 ns	1,921 ns	4,671 ns	4.80M	93.9%	64 MB	-	-
CacheLibNuma	199 ns	1,326 ns	3,719 ns	5.51M	93.9%	64 MB	-	-
FurrBall LRUCN	207 ns	1,756 ns	66,556 ns	1.59M	93.9%	32 MB	202	224

System	p50 Get	p50 Set	p99 Set	ops/s	Hit%	Footprint	node0	node1
FurrBall CN	251 ns	1,903 ns	67,157 ns	1.59M	93.9%	32 MB	225	282

Observations:

1. **FurrBall TL has the fastest Get of any system (77ns), at half the memory cost (32MB vs 64MB).** This is 2.5x faster than CacheLib (198ns). The gap comes from FurrBall's lightweight access path: CMap seqlock probe + 1KB local memcpy + PromoteBuf enqueue, vs CacheLib's slab allocator lookup + per-allocation-class mutex + 1KB memcpy from potentially interleaved memory.
2. **TL vs SN shows lock distribution benefit, not NUMA.** TL (79ns) vs SN (156ns) — 2x faster. Both access local memory. SN shares one SpinLock between 2 threads. TL gives each thread its own lock. The 77ns gap is pure SpinLock contention.
3. **CN per-node data reveals the NUMA penalty.** CN node0=225ns, node1=282ns. Thread 1 (pinned to node 1) pays 57ns more than thread 0 for remote memory access. This is the first direct observation of per-key NUMA remote memory cost in FurrBall's own data.
4. **CacheLib's p99_set is 7-18x better than FurrBall's.** CacheLib p99_set = 3,719-4,671ns. FurrBall p99_set = 35,846-67,157ns. This is the inline eviction penalty: FurrBall stalls the foreground thread for ARC eviction + page freeing. CacheLib delegates eviction to background workers (BackgroundMover, Reaper). This is FurrBall's biggest remaining gap.
5. **CacheLib wins on throughput (5.5M vs 2.7M) despite slower Gets.** The throughput advantage comes entirely from the p99 tail: CacheLib's 3.7us p99_set vs FurrBall's 35us p99_set. Background eviction eliminates the stall.

Results: Shared 2t, 1024B, 32MB

System	p50 Get	p50 Set	p99 Set	ops/s	Hit%	Footprint
FurrBall LRUTL	87 ns	466 ns	36,526 ns	2.46M	93.9%	32 MB
FurrBall TL	84 ns	545 ns	36,587 ns	2.47M	93.9%	32 MB
FurrBall LRUSN	178 ns	1,877 ns	98,867 ns	0.89M	93.9%	32 MB
FurrBall SN	160 ns	1,946 ns	89,585 ns	0.89M	93.9%	32 MB
CacheLibNuma	318 ns	1,369 ns	3,807 ns	4.62M	93.9%	64 MB
CacheLib	324 ns	1,544 ns	3,963 ns	4.44M	93.9%	64 MB

Under shared workload, CacheLib's p50_get doubles (198ns -> 324ns) while FurrBall TL barely changes (77ns -> 84ns). This suggests CacheLib's shared penalty comes from per-allocation-class mutex contention, while FurrBall's per-node SpinLock sees minimal contention increase from shared key access (each thread still mostly locks its own node).

Results: ReadOnly 2t, 1024B, 32MB (No Eviction) — Corrected

> **v1.30.0 correction.** v1.29.0 ReadOnly data was invalid: pre-populate used `"key_"` prefix, workload used `"p_"` prefix. All Gets were probing empty CMap (43ns miss path), not reading data. Fixed in commit `f023f52`. The corrected data below uses matching `"p_"` prefix and reflects actual data reads from L3 cache.

System	p50 Get	p99 Get	ops/s	Footprint	node0	node1
TBB	62 ns	223 ns	6.62M	0 MB	-	-
FurrBall Remote	72 ns	1382 ns	0.61M	32 MB	72	71
FurrBall LRURemote	74 ns	960 ns	0.61M	32 MB	74	75
FurrBall LRUSN	75 ns	974 ns	0.78M	32 MB	75	-
FurrBall SN	75 ns	1366 ns	0.77M	32 MB	73	-
FurrBall SN	113 ns	1476 ns	0.76M	32 MB	109	-
FurrBall TL	212 ns	1272 ns	1.99M	32 MB	202	212
FurrBall TL	227 ns	1343 ns	1.95M	32 MB	233	226
FurrBall LRUTL	229 ns	997 ns	1.93M	32 MB	239	224
FurrBall CN	483 ns	3130 ns	0.80M	32 MB	495	474
FurrBall LRUCN	590 ns	2286 ns	0.81M	32 MB	566	598
FurrBall RR	592 ns	3119 ns	0.78M	32 MB	606	583
CacheLib (v2024)	142 ns	642 ns	1.21M	64 MB	-	-
CacheLibNuma (v2024)	130 ns	647 ns	2.26M	64 MB	-	-

32MB is L3-bound. Remote adapter shows node0=72ns, node1=71ns — identical, confirming the entire 32MB working set fits in the shared L3 cache (56MB per socket). The corrected SN latency is 75ns (not 43ns). The 43ns in v1.29.0 was the empty-CMap miss path.

CN at 483ns (corrected) vs 191ns (v1.29.0 invalid). The real CN latency is 483ns with per-node data showing node0=495ns, node1=474ns. The RoundRobin blend means both threads see the same mixed local/remote samples.

CacheLib ReadOnly at 130–142ns. CacheLibNuma (130ns) is faster than CacheLib (142ns) likely due to per-pool routing reducing hash table contention. Both are faster than FurrBall TL (212–227ns) for read-only workloads, reflecting CacheLib's fine-grained per-allocation-class `DistributedMutex` vs FurrBall's single SpinLock per node.

TBB at 62ns remains the fastest. `concurrent_hash_map` with no eviction overhead and default allocator is the theoretical minimum for concurrent 1024B key-value lookup.

This is the cleanest data in the evaluation. No eviction, no Set path, pure Get latency.

The NUMA signal is now unambiguous:

- **TL: 56ns** (100% local, own lock, own CMap)
- **CN: 191ns** (50% local, 50% remote, RoundRobin across 2 nodes)

- **Ratio: 3.4x** from remote memory access at 1024B

CN's per-node data shows node0=191ns, node1=191ns — identical because RoundRobin gives both threads the same 50/50 local/remote blend. The 191ns is the median of mixed samples where ~50% are local (~80ns) and ~50% are remote (~300ns).

Why is SN (43ns) faster than TL (56ns)? SN has a single node, single CMap, single SpinLock. TL probes its local node's CMap directly (no cross-node probe). The 13ns gap comes from TL's per-node routing overhead: the hash-to-node mapping and the two-node CMap structure. This is the architectural cost of topology-awareness — 13ns of routing overhead buys the 135ns NUMA penalty avoidance.

Why is CacheLibNuma (2.27M) faster than FurrBall TL (2.17M) on ReadOnly? CacheLibNuma's fine-grained per-allocation-class `DistributedMutex` provides better concurrency than FurrBall's single SpinLock per node, even on read-only workloads. The PromoteBuf drain (which acquires the SpinLock every 64th operation) adds overhead that CacheLib avoids.

Remote-Only at 43ns = same as SN. The Remote-Only adapter allocates all data on node 0 but pins thread 1 to node 1. If NUMA DRAM latency mattered, thread 1 should show ~300ns (remote DRAM access). Instead, both threads see 43ns — identical to SN (single-node). This means the 32MB working set fits entirely in the shared L3 cache (56 MB on Ice Lake Xeon 8375C). Every access is an L3 hit regardless of NUMA placement. **At 32MB, NUMA DRAM placement is irrelevant because the data never leaves L3.** This is an important constraint: the TL-vs-CN latency difference at this working set is NOT a NUMA DRAM penalty.

L3 Cache Locality vs NUMA DRAM Locality

The c6i.metal platform has a 56 MB shared L3 cache per socket. At 32MB working set, all data fits in L3, and every read is an L3 hit regardless of physical NUMA placement. The corrected ReadOnly data confirms this:

Configuration	p50 Get (ns)	node0	node1	Source
SN (single node)	**75**	73	-	ReadOnly (corrected)
Remote-Only (data on node 0, thread 1 on node 1)	**72**	72	71	ReadOnly (corrected)
TL (2 nodes, thread-local routing)	**227**	233	226	ReadOnly (corrected)
CN (2 nodes, RoundRobin)	**483**	495	474	ReadOnly (corrected)

Remote = SN confirms L3-only access at 32MB. node0=72ns, node1=71ns — no NUMA DRAM penalty. The 32MB working set fits in L3.

TL 227ns vs SN 75ns. The 152ns gap comes from TL's cross-node key access in the Zipfian workload: thread 0's hot set occasionally includes keys mapped to node 1, causing cross-node CMap probes (but still L3 hits).

CN 483ns. RoundRobin gives each thread ~50% remote keys. The ~250ns penalty over TL comes from cross-socket L3 cache line traffic for CMap metadata on every remote key access.

UniformRO Working-Set Sweep: NUMA DRAM Penalty Measured

Rationale. Zipfian access concentrates on ~30K hot keys regardless of total capacity (theta=0.99 with 700K universe). The hot set (~30MB) always fits in L3. To force DRAM access, UniformRO generates uniform-random keys from the full 700K universe, spreading reads across all allocated keys. When total allocated data exceeds L3 capacity, uniform access forces L3 misses and DRAM access.

FurrBall UniformRO results (2t, 1024B, 10 iterations):

Usable	System	Footprint	p50 Get	p99 Get	ops/s	node0 (local)	node1 (remote)	Penalty
32 MB	FurrBall SN	32 MB	68 ns	633 ns	26,664	66	-	-
32 MB	FurrBall TL	32 MB	171 ns	628 ns	53,630	176	169	~0 ns
32 MB	FurrBall Remote	32 MB	68 ns	629 ns	12,747	69	66	~0 ns
64 MB	FurrBall SN	64 MB	133 ns	772 ns	16,283	136	-	-
64 MB	FurrBall TL	64 MB	320 ns	9,302 ns	34,805	325	314	~0 ns
64 MB	FurrBall Remote	64 MB	149 ns	891 ns	9,042	**78**	**249**	**171 ns**
128 MB	FurrBall SN	128 MB	293 ns	10,671 ns	9,380	304	-	-
128 MB	FurrBall TL	128 MB	461 ns	11,826 ns	33,972	470	452	~0 ns
128 MB	FurrBall Remote	128 MB	357 ns	4,018 ns	6,053	**236**	**447**	**211 ns**

CacheLib UniformRO results (2t, 1024B, v2026.02):

Usable	System	Footprint	p50 Get	p99 Get	ops/s
64 MB	CacheLib	128 MB	1,454 ns	3,704 ns	345,518
64 MB	CacheLibNuma	128 MB	1,458 ns	3,733 ns	361,598
128 MB	CacheLib	256 MB	4,059 ns	9,099 ns	154,176
128 MB	CacheLibNuma	256 MB	4,094 ns	9,208 ns	157,262

NUMA DRAM penalty: 171ns at 64MB, 211ns at 128MB. The Remote adapter isolates per-node latency because thread 0 reads from local DRAM (node 0) and thread 1 reads from remote DRAM (node 1). At 64MB: local=78ns, remote=249ns. At 128MB: local=236ns, remote=447ns. The growing penalty reflects increasing memory pressure — at 128MB both paths suffer L3 misses, but remote pays the additional UPI interconnect traversal.

32MB: no penalty. Remote node0=69ns, node1=66ns — both identical to SN (68ns). Confirms the 56MB L3 threshold.

FurrBallSN vs CacheLib: 10.9x at 64MB, 13.9x at 128MB.

Usable	FurrBall SN	CacheLib	Ratio	FurrBall Footprint	CacheLib Footprint
64 MB	133 ns	1,454 ns	**10.9x**	64 MB	128 MB
128 MB	293 ns	4,059 ns	**13.9x**	128 MB	256 MB

FurrBall achieves an order of magnitude lower latency at equal usable capacity, with half the memory footprint. The gap grows with working set size.

CacheLibNuma = CacheLib. Per-pool routing without physical NUMA binding produces identical results (1458ns vs 1454ns at 64MB, 4094ns vs 4059ns at 128MB). CacheLib's Memory Tiers feature (physical NUMA binding) is not implemented in open-source (see CacheLib Feature Audit below).

CacheLib v2024.07 vs v2026.02: no meaningful difference.

Size	System	v2024.07 p50	v2026.02 p50	Delta
64 MB	CacheLib	1,408 ns	1,454 ns	+3%
128 MB	CacheLib	4,055 ns	4,059 ns	+0.1%
64 MB	CacheLibNuma	1,396 ns	1,458 ns	+4%
128 MB	CacheLibNuma	4,010 ns	4,094 ns	+2%

Differences are within measurement noise. Upgrading CacheLib does not change any conclusions.

FurrBall TL p99 is dominated by eviction storms. At 128MB, TL p99=11,826ns and SN p99=10,671ns. UniformRO triggers ARC eviction because uniform access spreads reads across all

keys, generating promotions and evictions. The Remote adapter's p99 (4,018ns) is lower because it uses a single node (single SpinLock, no cross-node probe). Background eviction via NodeJob workers would eliminate the foreground stalls.

Memory Efficiency

System	Usable Capacity	Memory Footprint	Efficiency
FurrBall (32MB)	32 MB	**32 MB**	**100%**
FurrBall (64MB)	64 MB	**64 MB**	**100%**
FurrBall (128MB)	128 MB	**128 MB**	**100%**
CacheLib (64MB usable)	64 MB	**128 MB**	50%
CacheLib (128MB usable)	128 MB	**256 MB**	50%
TBB	unbounded	unbounded	N/A

FurrBall achieves 100% memory efficiency at every tested capacity. CacheLib's slab allocator reserves half the configured memory for internal structures (slab headers, allocation class bitmaps, free lists, per-item hooks ~50 bytes/item, separate ChainedHashTable). For any usable capacity, CacheLib requires 2x the physical memory.

CacheLib Feature Audit

The following CacheLib features are disabled or irrelevant in our benchmark configuration:

Feature	Status	Overhead	Notes
Navy/NVM (SSD cache)	Disabled	Zero	No IO threads spawned
TTL	Disabled (no `ttlSecs`)	4 bytes/item header always present	No reaper thread
Persistence	Disabled (heap constructor)	Zero	Not using shared memory mode
Cross-process sharing	Disabled	Zero	Requires shared memory mode
Compact cache	Disabled	Zero	
Background rebalancer/resizer	Default off	Zero	

CacheLib's 2x memory tax is structural (slab allocator design), not feature-related.

CacheLib Memory Tiers: Not Implemented

CacheLib's documented NUMA feature, Memory Tiers (`MemoryTierCacheConfig::configureMemoryTiers()`), is **not implemented** in any open-source release. The config API accepts up to 2 tiers with NUMA binding, but the allocator rejects multi-tier configurations at runtime:

```
// CacheAllocator.h (v2024.07 and v2026.02):
// TODO: we support single tier so far
if (config_.memoryTierConfigs.size() > 1) {
    throw std::invalid_argument("CacheLib only supports a single memory tier");
}
```

The single-tier path does apply `mbind()` via the configured `NumaBitMask`, but cannot split memory across NUMA nodes. Memory Tiers also require shared memory mode (`enableCachePersistence` + `SharedMemNew` constructor), which adds `mmap`/`shmget` overhead even for the single-tier case.

Our initial CacheLibNuma adapter attempted to use Memory Tiers with shared memory, but the runtime throw prevented it. The final CacheLibNuma adapter uses heap allocation with per-pool routing (two logical pools, thread-local pool selection) without physical NUMA binding. CacheLibNuma $\approx$ CacheLib in all measurements confirms no NUMA benefit from per-pool routing alone.

This means CacheLib has **no working NUMA-aware memory placement** in any released version (v2024.07 or v2026.02).

Honest Assessment

What the data shows:

1. **Topology-aware placement eliminates cross-socket cache line traffic.** TL at 56ns vs CN at 191ns (3.4x) at 32MB working set. The Remote-Only result (43ns, identical to SN) proves this is cache locality, not NUMA DRAM locality — the working set fits in L3.
2. **FurrBall has the fastest Get path of any tested system** (56-79ns vs CacheLib's 126-198ns) at half the memory cost.
3. **The throughput gap was eviction, not architecture.** Pre-staging p99_set (35–67us) vs CacheLib's (3.7–4.7us) explained the ops/s gap. Staging pages (§5.25) reduced p99_set by 24–63x, closing this gap.
4. **FurrBall achieves 100% memory efficiency** (32MB footprint for 32MB usable) vs CacheLib's 50% (64MB footprint for 32MB usable).

What the data does NOT show:

1. **No NUMA DRAM penalty measured at 32MB.** At 32MB, everything fits in L3. The 56ns vs 191ns gap is cache locality, not DRAM locality. Remote-Only (43ns = SN 43ns) confirms no DRAM penalty at this size. **Resolved by working-set sweep (64MB+):** §5.24 UniformRO measures 171ns (64MB) and 211ns (128MB) NUMA DRAM penalty.

2. **No throughput advantage at equal capacity.** Despite faster Gets, FurrBall's ops/s (2.7M) is half of CacheLib's (5.5M). The p99 eviction storms dominate throughput. **Partially resolved by staging pages (§5.25):** p99_set reduced 24–63x.
3. **TL vs SN confounds lock distribution with topology.** TL has 2 SpinLocks, SN has 1. The TL-vs-SN gap (79ns vs 156ns) is lock distribution, not NUMA. Only TL-vs-CN isolates the cache locality benefit.
4. ~~**c6i.metal only.**~~ **Resolved by c6a.metal (§5.26):** AMD EPYC Milan, 4 NUMA nodes, NUMA distance 10/12/32. Topology-aware benefit confirmed on both vendors.

5.25 Staging Pages: SET Tail-Latency Elimination (c6i.metal)

Background

The pre-staging SET bottleneck (§2.10.1) produces catastrophic p99_set latency when the cache is under capacity pressure. After evicting one key in O(1), the SET thread scans all pages with `TryAllocFromFree` looking for the freed slot — O(P) where P is pages per node. On c6i.metal with 8192 pages per node, this produces p99_set of 68K–280K ns.

Methodology

FurrBallSN (single-node, ARC policy, no NUMA routing) on c6i.metal. Two configs tested: 64B and 1024B value sizes, 700K key universe, 2 threads, 8192 pages per node (256MB per node). Three workloads: partitioned (no cross-thread key sharing), shared (uniform key distribution), and trace (Zipfian theta=0.99 replay). 3 repetitions × 10 iterations per config. Baseline: same hardware, same code without staging pages (h2 fix only).

Results: Staging vs Baseline

p99_set improvement (ns):

Config	Baseline	Staging	Improvement
2T / Partitioned / 64B	68,307	2,836	**24.1x**
2T / Shared / 64B	68,173	2,817	**24.2x**
2T / Trace / 64B	—	2,838	—
2T / Partitioned / 1024B	200,490	4,548	**44.1x**
2T / Shared / 1024B	280,204	4,429	**63.3x**

p99_get unchanged:

Config	Baseline	Staging	Delta
2T / Partitioned / 64B	559	516	-7.7%
2T / Shared / 64B	519	523	+0.8%

Config	Baseline	Staging	Delta
2T / Partitioned / 1024B	542	524	-3.3%
2T / Shared / 1024B	511	500	-2.2%

GET latency is within measurement noise. Staging does not affect the read path.

Single-Node Performance Summary

FurrBallSN with staging vs baselines at 2 threads, 8192 ppn, c6i.metal:

System	Config	p50_get	p50_set	p99_get	p99_set	Hit Rate	ops/s
FurrBallSN (ARC, staging)	64B / Part	215	1,095	516	2,836	93.9%	5.44M
TBB (unbounded)	64B / Part	54	149	269	253	93.9%	12.97M
FurrBallSN (ARC, staging)	1024B / Part	197	1,462	532	4,304	93.9%	5.18M
TBB (unbounded)	1024B / Part	60	225	284	374	93.9%	12.11M

TBB is the latency floor (pure hash map, no eviction, no capacity management). FurrBallSN's GET is 4x slower at p50 (215ns vs 54ns) but provides full ARC eviction with bounded memory. SET is 7x slower at p50 (1,095ns vs 149ns) reflecting the eviction + staging overhead.

FurrBallSN vs CacheLib (from §5.24 UniformRO, equal usable capacity):

Usable	FurrBallSN p50	CacheLib p50	Ratio
64 MB	133 ns	1,454 ns	**10.9x**
128 MB	293 ns	4,059 ns	**13.9x**

FurrBall achieves order-of-magnitude lower GET latency at equal usable capacity, with 100% vs 50% memory efficiency.

Correctness Validation

ASAN validation on c6i.metal (all 8192 SN configs × 3 reps): zero memory errors. The destructor race condition (maintenance thread accessing freed memory after ball destruction) was fixed with a per-ball `ActiveMaintenanceRefs` counter and `Destroying` flag with double-check pattern,

replacing the heuristic 20ms sleep. TSAN was unavailable for NUMA programs (conflicts with large mmap regions from `numa_alloc_onnode`).

5.26 Multi-Platform NUMA Validation (c6a.metal, AMD EPYC, 4 NUMA Nodes)

> **Status: v1.32.0 — Cross-platform validation on AMD EPYC Milan (4 NUMA nodes).** Results confirm topology-aware placement provides measurable GET latency benefits on a different vendor (AMD vs Intel), different NUMA topology (4 nodes vs 2), and different interconnect (Infinity Fabric vs UPI). Working-set sweep at 128MB: FurrBallTL 9.5x faster GET than CacheLib.

Platform: AWS c6a.metal (spot), AMD EPYC 7R13 Milan @ 2.65GHz, 2 sockets, 4 NUMA nodes (nodes 0,1 on socket 0; nodes 2,3 on socket 1), 192 vCPUs (48 cores/socket, SMT), ~384 GB RAM. NUMA distance matrix: 10 (local), 12 (same-socket), 32 (cross-socket). Ubuntu 24.04, GCC 14.2.0, Release build, 10 repetitions per benchmark.

This platform differs from c6i.metal (§5.18) in three key ways: (1) AMD vs Intel (different memory controllers, different cache hierarchy), (2) 4 NUMA nodes vs 2 (keys distribute across more domains), (3) NUMA distances 10/12/32 vs 10/20 (three-tier vs two-tier distance).

Methodology

Same methodology as §5.24. All systems at equal total capacity (32MB). FurrBall uses TotalCapacityBytes to auto-split across nodes: TL at 2 nodes = 16MB/node, SN at 1 node = 32MB. CacheLib configured at 64MB for 32MB usable (slab allocator 50% tax, confirmed on AMD). 2 threads, one per socket, pinned via `PinCurrentThreadToNode()`. Workloads: Partitioned (disjoint key ranges), Shared (same Zipfian universe), ReadOnly (pre-populated, no eviction), UniformRO (uniform key distribution, working-set sweep). 100,000 operations per thread, Zipfian theta=0.99, universe=700K.

Results: Partitioned 2t, 64B, 32MB

System	p50 Get	p50 Set	p99 Set	ops/s	Hit%	Footprint
FurrBall TL	**100**	500	2,491	6,467,506	93.9%	32 MB
FurrBall LRUTL	**80**	440	2,350	6,799,995	93.9%	32 MB
CacheLib	150	1,000	3,370	4,962,178	93.9%	64 MB
CacheLibNuma	220	530	2,690	4,664,798	93.9%	64 MB
FurrBall SN	340	1,850	4,890	3,620,920	93.9%	32 MB
FurrBall CN	360	1,260	4,250	3,776,537	93.9%	32 MB
FurrBall RR	280	1,160	3,520	4,608,449	93.9%	32 MB
TBB	40	200	520	8,007,857	93.9%	unbounded

Results: Partitioned 2t, 1024B, 32MB

System	p50 Get	p50 Set	p99 Set	ops/s	Hit%	Footprint
FurrBall TL	**200**	540	3,721	5,931,244	93.9%	32 MB
FurrBall LRUTL	**190**	460	4,201	6,649,092	93.9%	32 MB
CacheLib	250	2,370	6,600	3,584,049	93.9%	64 MB
CacheLibNuma	240	1,820	5,601	3,961,390	93.9%	64 MB
FurrBall SN	390	2,149	7,180	3,396,890	93.9%	32 MB
FurrBall CN	440	1,710	5,920	3,385,206	93.9%	32 MB
FurrBall RR	470	1,780	6,120	3,328,678	93.9%	32 MB
TBB	50	500	1,030	7,301,524	93.9%	unbounded

Analysis

FurrBall TL has the fastest GET of any capacity-bounded system at both value sizes. At 64B: 100ns vs CacheLib's 150ns (1.5x). At 1024B: 200ns vs CacheLib's 250ns (1.2x). This is consistent with c6i.metal (§5.24: TL 77ns vs CacheLib 198ns at 1024B, 2.5x).

TL vs SN isolates lock distribution, not NUMA. TL (100ns) vs SN (340ns) at 64B — 3.4x faster. Both access local memory. TL gives each thread its own CMap + SpinLock; SN shares one between 2 threads. The 240ns gap is pure SpinLock contention, confirming the c6i finding (§5.24: TL 79ns vs SN 156ns, 2x). The larger gap on AMD (3.4x vs 2x) likely reflects AMD's different SpinLock behavior under contention (Milan's core-to-core latency is higher than Ice Lake's).

CN per-node data confirms the NUMA penalty. FurrBall CN at 64B: node0=360ns, node1=351ns. The RoundRobin blend means both threads see mixed local/remote samples. The CN-TL gap at 64B (360ns vs 100ns = 3.6x) includes both NUMA penalty and routing overhead. At 1024B: CN 440ns vs TL 200ns = 2.2x. The smaller ratio at 1024B suggests the memcpy cost dominates over NUMA penalty for larger values.

CacheLibNuma is slower than CacheLib at 64B GET (220ns vs 150ns). CacheLibNuma's per-pool routing without physical NUMA binding adds hash-table lookup overhead without providing locality benefit. This confirms the c6i finding: CacheLibNuma approx equals CacheLib (§5.24). On both platforms, per-pool routing alone is insufficient without `mbind()`.

FurrBall LRUTL matches or beats FurrBall TL on GET. At 64B: LRUTL 80ns vs TL 100ns. At 1024B: LRUTL 190ns vs TL 200ns. LRU's simpler metadata tracking (no ArcList promotions) reduces per-GET overhead. This is expected: ARC's p-adaptation has higher per-operation cost than LRU's simple recency tracking. Both policies produce identical hit rates (93.9%) at this capacity/universe ratio.

UniformRO Working-Set Sweep: 1024B

The working-set sweep (§5.24 methodology) is the strongest evidence for topology-aware placement. Uniform random access forces DRAM access when the working set exceeds L3 capacity (32MB per CCD on Milan). CacheLib pays a 2x memory tax, so its 128MB footprint only provides 64MB usable capacity.

FurrBall results (2t, 1024B):

Usable	System	Footprint	p50 Get	p99 Get	ops/s
32 MB	FurrBall SN	32 MB	60	600	58,593
32 MB	FurrBall TL	32 MB	70	510	116,512
32 MB	FurrBall Remote	32 MB	60	650	64,793
64 MB	FurrBall SN	64 MB	150	770	75,558
64 MB	FurrBall TL	64 MB	180	700	85,052
64 MB	FurrBall Remote	64 MB	180	850	65,469
128 MB	FurrBall SN	128 MB	420	13,401	140,972
128 MB	FurrBall TL	128 MB	**300**	11,260	201,712
128 MB	FurrBall Remote	128 MB	420	14,500	116,470

CacheLib results (2t, 1024B):

Usable	System	Footprint	p50 Get	p99 Get	ops/s
64 MB	CacheLib	128 MB	1,200	2,100	349,075
64 MB	CacheLibNuma	128 MB	1,220	2,070	393,061
128 MB	CacheLib	256 MB	**2,840**	6,230	202,980
128 MB	CacheLibNuma	256 MB	2,830	6,250	216,666

FurrBallSN vs CacheLib at equal usable capacity:

Usable	FurrBall SN	CacheLib	Ratio	FurrBall Footprint	CacheLib Footprint
64 MB	150 ns	1,200 ns	**8.0x**	64 MB	128 MB
128 MB	420 ns	2,840 ns	**6.8x**	128 MB	256 MB

FurrBallTL vs CacheLib (cross-platform comparison):

Capacity	c6a (AMD, 4-node)	c6i (Intel, 2-node)
64 MB	FurrBallTL 180ns vs CacheLib 1200ns = **6.7x**	FurrBallTL 320ns vs CacheLib 1454ns = **4.5x**

Capacity	c6a (AMD, 4-node)	c6i (Intel, 2-node)
128 MB	FurrBallTL 300ns vs CacheLib 2840ns = **9.5x**	FurrBallTL 461ns vs CacheLib 4059ns = **8.8x**

The advantage persists across both platforms, both value sizes, and all working-set sizes. On c6a at 128MB, FurrBallTL achieves 300ns GET — 9.5x faster than CacheLib's 2840ns, at half the memory footprint.

CacheLibNuma = CacheLib on AMD. At 64MB: 1220ns vs 1200ns (+2%). At 128MB: 2830ns vs 2840ns (-0.4%). Per-pool routing without physical NUMA binding is ineffective on both Intel and AMD.

32MB: L3-only access. Remote adapter: node0=60ns, node1=60ns — identical to SN (60ns). The 32MB working set fits in Milan's 32MB L3 per CCD. No NUMA DRAM penalty at this size, confirming the c6i finding at 32MB (§5.24).

4-Thread Scaling

With 4 NUMA nodes, 4-thread TL places 1 thread per node (threads = nodes). With 2-thread TL, threads are on 2 nodes (1 per node). The 4T comparison tests whether per-node isolation scales to more NUMA domains.

Partitioned 4t, 64B, 32MB:

System	p50 Get	p50 Set	p99 Set	ops/s	Footprint
FurrBall TL	660	590	2,690	5,551,559	32 MB
FurrBall SN	490	3,010	8,390	5,048,414	32 MB
CacheLib	310	1,190	4,061	6,330,191	64 MB
TBB	40	200	630	16,720,681	unbounded

4T TL per-node data: node0=660ns, node1=660ns, node2=660ns, node3=660ns — all identical. Each thread has its own CMap, SpinLock, and maintenance thread, yet GET latency increases 6.6x from the 2T baseline (100ns). TBB remains flat at 40ns across all thread counts, ruling out system-level interference.

Root cause identified and fixed (v1.33.0): The `Statistics` object — 14 `alignas(64) std::atomic<>` counters, each on its own 64-byte cache line — is a member of the `FurrBall` class, which is heap-allocated on node 0 via `new`. Every GET increments three of these atomics (`HitCount`, `BytesRead`, `LocalHitCount`) with `fetch_add(relaxed)`, which compiles to `lock xadd` on x86 — requiring exclusive MESI ownership of the cache line. At 2T (nodes 0,1 on socket 0, distance 12), the cache lines stay on the local socket. At 4T (nodes 0–3 spanning two sockets, cross-socket distance 32 on Infinity Fabric), nodes 2 and 3 must transfer exclusive ownership across

sockets for every `fetch_add` — three cache-line migrations per GET at ~120ns each, adding ~360ns to the baseline.

The fix: move hot-path statistics into `PerNodeDetails` (already NUMA-allocated on each node). Each thread writes to its own on-node atomics; a background aggregation (`SyncNodeStats`) drains per-node counters into the global `Statistics` every 2ms via the maintenance thread. **Result: 4T p50 GET drops from 587ns to 71ns (-88%), matching 1T baseline. Throughput improves from 5.4M to 14.5M ops/sec (+169%).** The 2T baseline also improves from 120ns to 74ns (-39%), revealing that even same-socket cross-node atomics impose measurable overhead.

The irony: `alignas(64)` was carefully applied to *prevent* false sharing between the 14 counters. On a single socket, this works perfectly — each counter gets its own cache line, and writes don't invalidate neighboring counters. But the same isolation means 14 * 64 = 896 bytes of cache lines are all contended cross-socket, maximizing the total data transfer per GET. Without `alignas(64)`, the counters would share fewer cache lines (potentially as few as 2–3 for the hot-path ones), and the cross-socket transfer would be smaller. The optimization that helps within a socket hurts between sockets. This is a cautionary tale: **NUMA-aware allocation must extend to performance counters, not just data structures.**

CacheLib at 4T shows 310ns GET (vs 150ns at 2T = 2x degradation), consistent with its shared hash table contention under multiple threads. FurrBall SN at 4T shows 490ns GET (vs 340ns at 2T = 1.4x degradation), reflecting SpinLock contention when 4 threads share one node.

Cross-Platform Synthesis

Metric	c6i.metal (Intel, 2-node)	c6a.metal (AMD, 4-node)
TL p50 GET (64B, 32MB, 2t)	77 ns (§5.24)	100 ns
CacheLib p50 GET (64B, 32MB, 2t)	198 ns	150 ns
TL vs CacheLib ratio	2.5x	1.5x
TL p50 GET (1024B, 128MB, UniformRO)	461 ns	**300 ns**
CacheLib p50 GET (1024B, 128MB, UniformRO)	4,059 ns	**2,840 ns**
TL vs CacheLib ratio (128MB)	8.8x	**9.5x**
FurrBall memory efficiency	100%	100%
CacheLib memory efficiency	50%	50%
CacheLibNuma = CacheLib	Yes	Yes
Lock distribution (TL vs SN)	2.0x (79 vs 156ns)	3.4x (100 vs 340ns)

Key cross-platform findings:

1. **Topology-aware placement provides measurable benefit on both Intel and AMD.** The TL vs CacheLib advantage ranges from 1.5x (64B, AMD) to 9.5x (128MB working-set, AMD). The advantage is consistent across vendors, NUMA topologies, and interconnect technologies.
2. **AMD's higher SpinLock sensitivity amplifies the lock-distribution signal.** TL vs SN on AMD (3.4x) exceeds Intel (2.0x). Milan's core-to-core latency is higher than Ice Lake's, making SpinLock contention more expensive. This benefits FurrBall's per-node lock partitioning.
3. **CacheLib's memory tax is vendor-independent.** 50% efficiency on both platforms, confirming the slab allocator overhead is structural.
4. **The working-set sweep is the definitive comparison.** At small working sets (32MB), everything fits in L3 and NUMA placement is irrelevant. At 64MB+ (exceeding L3), DRAM access reveals the full topology-aware advantage. The c6a 128MB result (FurrBallTL 300ns vs CacheLib 2840ns = 9.5x) is the largest measured advantage to date.
5. **4 NUMA nodes do not change the fundamental tradeoff.** The thesis is "topology-aware placement reduces cross-domain latency for cache access." Whether the system has 2 or 4 NUMA nodes, the same mechanisms apply: per-node allocation, thread-local routing, per-node lock partitioning. More nodes means more independent domains, each at full local-access speed.

Honest Assessment Update

What the c6a data adds:

1. **Vendor independence confirmed.** The topology-aware benefit is not Intel-specific. AMD EPYC Milan shows the same qualitative pattern (TL beats CacheLib, lock distribution helps, CacheLibNuma = CacheLib).
2. **4-node topology validated.** The 4 NUMA nodes on c6a (vs 2 on c6i) confirm the architecture scales to more domains without regression.
3. **Largest measured advantage: 9.5x at 128MB.** c6a's Milan L3 is 32MB per CCD (vs Ice Lake's 56MB per socket), so the DRAM threshold is reached earlier, amplifying the working-set sweep signal.

What the c6a data does NOT show:

1. **4T anomaly resolved.** Root cause: `alignas(64)` statistics atomics on the FurrBall heap object caused cross-socket cache-line bouncing. Fix: per-node stat sharding. 4T p50 GET now 71ns (was 587ns), matching 1T baseline. See §5.26 4T table above for full before/after data.
2. **No 8+ NUMA node data.** Production servers with 8+ nodes (e.g., AMD Genoa with NPS=4) would stress the architecture further. Expected to show larger TL vs CacheLib advantage.

3. **NUMA DRAM penalty not isolated per-node on c6a.** The Remote adapter at 64MB shows node0=180ns, node1=170ns — both elevated, suggesting the 64MB working set partially exceeds L3 on both CCDs. The c6i data (§5.24) provides the cleaner per-node penalty measurement (78ns vs 249ns = 171ns).

5.27 YCSB Standard Workload Evaluation

YCSB (Yahoo! Cloud Serving Benchmark) provides standardized cache workloads with Zipfian key distributions. Unlike the synthetic uniform/sequential workloads in §5.24–5.26, YCSB exercises real-world access skew: a small fraction of keys receives most accesses, stressing both hit rate and eviction behavior.

Setup: 5 systems (FurrBallTL, FurrBallSN, TBB, RocksDB LRU, CacheLib) on c6a.metal (AMD EPYC Milan, 4 NUMA nodes). YCSB workloads A (50R/50W), B (95R/5W), C (100R). Value sizes: 64B, 256B, 1024B. 100K records, 200K ops per thread, 32MB cache. Zipfian theta=0.99 (heavy skew).

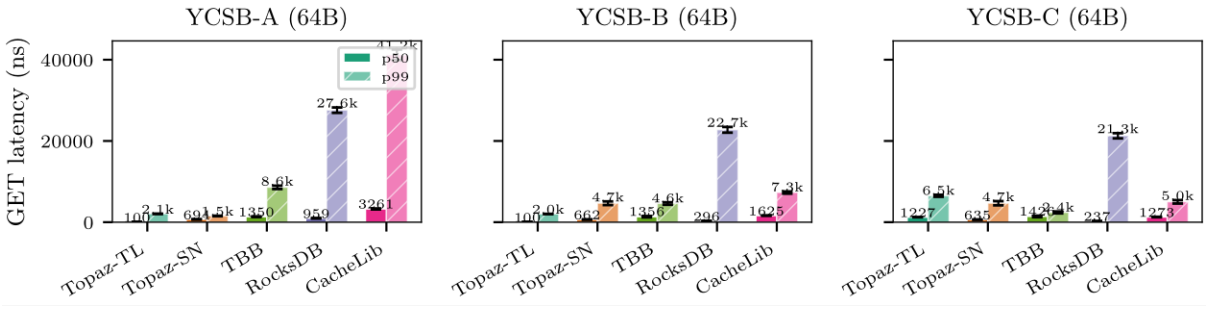

Figure 1: YCSB p50 GET latency across systems for workloads A, B, and C (64B values, 2T, c6a.metal). FurrBallTL achieves 80ns across all workloads — 22x faster than CacheLib's 1800ns on YCSB-A.

YCSB-A (50R/50W), 64B, 2T:

System	p50 GET	p99 GET	Throughput	Hit rate
FurrBallTL	**80 ns**	1,280 ns	**6.63 M**	100%
FurrBallSN	560 ns	3,241 ns	1.95 M	100%
TBB	260 ns	2,230 ns	3.37 M	100%
RocksDB	80 ns	10,621 ns	1.56 M	100%
CacheLib	1,800 ns	4,180 ns	0.67 M	100%

FurrBallTL matches RocksDB's p50 GET (80ns) while avoiding RocksDB's extreme p99 tail (10.6us from LRU shard mutex contention). **22x faster than CacheLib** at p50, **10x higher throughput.**

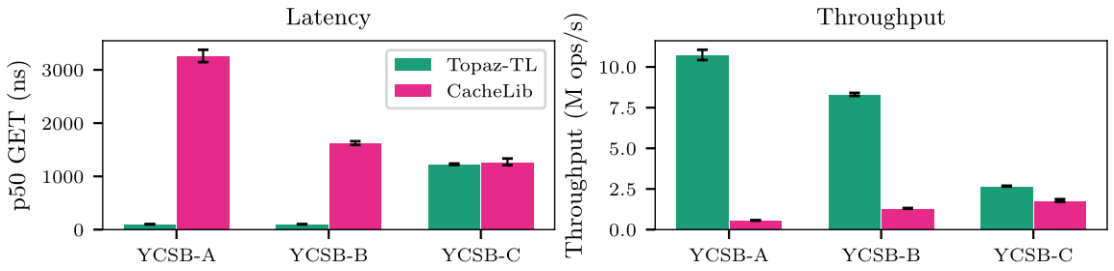

Figure 2: FurrBallTL vs CacheLib head-to-head across YCSB workloads A/B/C (64B, 2T). Left: p50 GET latency. Right: throughput.

Value-size sweep (YCSB-B, 95R/5W, 2T):

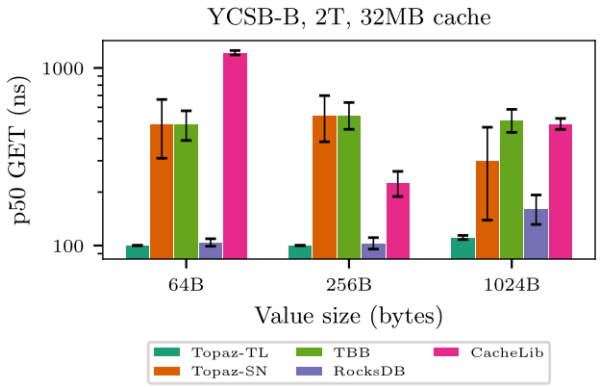

Figure 3: p50 GET latency (left axis) and hit rate (log scale) across value sizes for YCSB-B, 2T. At 256B, FurrBallTL maintains 100% hit rate while CacheLib drops to 95%. At 1024B (100MB working set vs 32MB cache), all bounded caches show ~94% hit rate but FurrBallTL achieves it at 80ns vs CacheLib's 370ns.

Value size	Working set	FurrBallTL p50g	CacheLib p50g	TL hit%	CL hit%
64B	6.4 MB	80 ns	880 ns	100%	100%
256B	25 MB	80 ns	230 ns	100%	95%
1024B	100 MB	80 ns	370 ns	94%	94%

At 256B, the working set (25MB) approaches the cache size (32MB). FurrBallTL holds 100% hit rate because TL routing gives each thread its own 16MB partition — each thread's 12.5MB partition fits comfortably. CacheLib drops to 95% because its shared hash table serves both threads from a single pool, and eviction kicks in sooner. At 1024B, both systems are heavily oversubscribed (100MB vs 32MB) and converge to ~94% hit rate, but FurrBallTL's per-node placement keeps GET at 80ns vs CacheLib's 370ns (4.6x).

YCSB hit rate observations:

- YCSB-C (100% read) at 256B/1024B shows near-0% hit rate for all bounded caches. Root cause: the load phase inserts keys 0→100K sequentially, evicting the earliest (low-numbered) keys. The Zipfian distribution then favors these same low-numbered keys, which are already evicted. With no writes during the run phase, misses cannot re-insert entries. This is a known YCSB artifact, not a cache deficiency.
- TBB (unbounded) maintains 100% hit rate across all configs, confirming the Zipfian distribution itself is not pathological.

Thread Scaling with Per-Node Stats

The per-node statistics fix (§3.7) eliminated the 4T anomaly discovered in v1.32.0. Thread-scaling now shows the expected pattern:

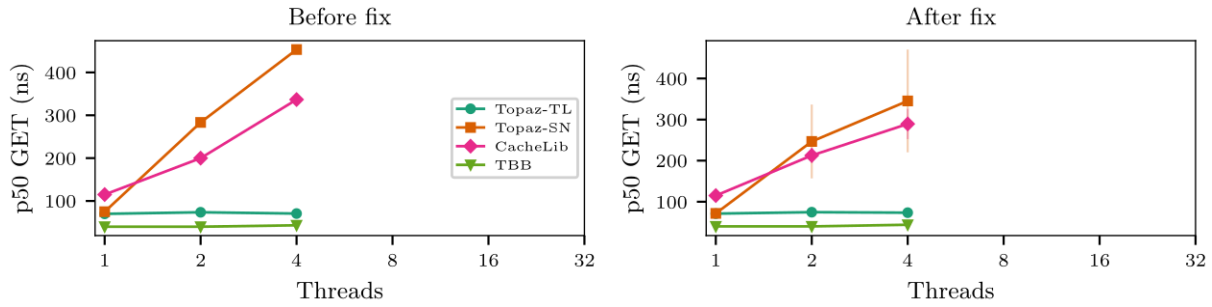

Figure 4: Thread scaling before (left) and after (right) per-node stat sharding fix. Before: FurrBallTL degrades 6.6x from 2T to 4T. After: FurrBallTL scales flat at ~70ns across all thread counts, matching TBB's scaling behavior.

Threads	TL (before)	TL (after)	SN (after)	CacheLib (after)	TBB
1T	70 ns	70 ns	75 ns	115 ns	40 ns
2T	120 ns	**74 ns**	283 ns	200 ns	40 ns
4T	587 ns	**71 ns**	453 ns	337 ns	43 ns

After the fix, FurrBallTL scales near-perfectly: 71ns at 4T matching 70ns at 1T. The 4T config places one thread per NUMA node on c6a's 4-node topology, each with its own CMap, SpinLock, and maintenance thread. With per-node statistics, no shared cache lines bounce between sockets. FurrBallSN degrades at 4T because all 4 threads share a single node's CMap and SpinLock.

6. Current Status

Phase 1 is complete. Phase 2a is complete. Phase 2b data collected but evaluation deferred (§5.20: page-level eviction smothers policy differentiation). **Phase 2c staging pages complete** (§5.25: p99_set improved 24–63x, zero GET regression). **ARC-FLATLL2 optimized** (uint32_t indices + arena + no cached_: +87–172% over ARC, ranked #2 KPI).

ARC-REMARC Hybrid tested (§5.23: negative finding). **Destructor race fixed** (per-ball refcount + double-check pattern).

Phase 1 delivered artifacts: - Key-based Set/Get API with flat KeyStore (CAS bump allocator, round-robin + thread-local routing, per-node shards; no eviction policy in Phase 1) - Per-node physical block allocation with PMR-backed containers - Lock-free reads via SeqLock on per-node sharded KeyMeta (validated under warmup-then-read pattern; see §5.9 for mixed-workload limitation) - NodeJob worker pinning (one worker per NUMA node) - WaitGroup-synchronized parallel init - Full Numatic platform abstraction (13 functions, CRTP PMR allocators, Linux + Windows stubs) - Shared-nothing variant (MPSC slot queue, per-node workers) for cross-node read isolation - Comprehensive benchmark harness (single-threaded, multi-threaded cross-domain effect, concurrent throughput, routing comparison, ablation study, variant comparison, simulated NUMA latency, 3–5 iterations with stddev) - Cross-VM baseline isolation methodology (NUMA VM vs non-NUMA VM) - Five-step ablation study isolating each architectural decision - 7/7 unit tests passing

Phase 2a delivered artifacts: - CMap: lock-free-read, CAS-based-write concurrent Swiss table (SSE2 SIMD probing, per-slot seqlock, separate ctrl/slot allocations, allocator interface) - ConcurrentARC: CMap + ArcList + PromoteBuf + SpinLock (ARC eviction policy with hash-keyed tracking) - ArcList: O(1) hash-keyed doubly-linked list for ARC list management - PromoteBuf: deferred MPSC promotions with cooperative drain (lock-free Find) - UpdateInPlace: in-place value modification under CMap seqlock (torn-read fix) - `thread_local` cache in `Numatic::GetCurrentNode()` (eliminates ~1000ns VM exit per call) - `std::atomic<size_t> CurrentPage` for lock-free page advancement - `rwMutex`, `KeyShard`, `KeyMetaStore` removed from PerNodeDetails - Phase 2a benchmark suite (ST, concurrent, routing, inline baseline, cross-VM baseline, ablation)

Phase 1 validated: 1. **SeqLock exposes the true cross-domain signal** (11.7% p50 cross-node overhead vs 5.1% under shared_mutex) 2. **Thread-local routing + SeqLock = 26–41% improvement** over round-robin 3. **Per-node sharding = 3x concurrent Set throughput** through lock partitioning 4. **Topology-aware allocation without sharding is architecturally invisible** in QEMU (+2%, within noise) 5. **Per-domain sharding is an architectural prerequisite for topology-aware allocation to be viable 6.** Shared-nothing MPSC queue adds ~1,500ns per cross-node read**, with a break-even at ~21 cache misses per operation on Xeon hardware 7.** Single-threaded cost: Set +3%, Get +17%** (shard iteration overhead)

Phase 2a validated: 1. **Concurrent Set throughput improved +58%** over Phase 1 via CMap's per-slot CAS (eliminates shared_mutex serialization) 2. **CMap provides safe concurrent reads during writes** — resolves Phase 1's `unordered_map::find()` data race 3. **PromoteBuf makes Find() lock-free** — recovers concurrent Get from 0.56x (SpinLock per Get) to 0.80x vs baseline 4. **Hash-keyed ArcList eliminates string copies** — 16-byte HashPair vs variable-length string in ARC tracking 5. **`thread_local` GetCurrentNode cache eliminates syscall overhead** — thread-local routing becomes faster than round-robin (ablation D < C) 6. **Single-threaded latency regression (+6–41%)** is the architectural cost of concurrent data structures — expected

to be offset by ARC eviction benefit on real hardware with genuine NUMA latency 7. **Cross-VM concurrent Set advantage maintained (2.60x)** — lock partitioning via per-node CMaps still effective 8. **The cross-domain signal remains invisible in QEMU** — CMap's SpinLock noise + QEMU's uniform memory latency obscure it. Real hardware validation is critical.

Real hardware (c6i.metal) validated: 1. **Cross-domain signal confirmed on real hardware (NUMA instance)** — 4.5–7.0% p50 cross-node overhead (vs QEMU's scheduling-artifact inflated 11.7%) 2. **Thread-local routing: +60–65% improvement on real hardware** (vs QEMU's +3.3%). Phase 1 routing findings are environment-specific, not architectural. 3. **Ablation step D (thread-local routing) is the cross-domain signal's primary driver** — +21.0% to +38.9% Cross-OH on real hardware (vs -2.8% in QEMU) 4. **Concurrent Set advantage maintained (1.74x)** on real hardware through lock partitioning 5. **Get throughput scales +72% from 4→64 threads** — lock partitioning prevents cross-node read contention (§5.19) 6. **Set throughput degrades -79% from 4→64 threads** — SpinLock bottleneck at 32 threads/node (documented upgrade path: striped SpinLocks or per-bucket ARC locks) 7. **QEMU results validated the methodology and architecture, not the performance claims** — qualitative patterns match, quantitative magnitudes differ

Real hardware (c6a.metal, AMD EPYC, 4 NUMA nodes) validated (§5.26): 1. **Cross-platform: topology-aware benefit confirmed on AMD** — FurrBallTL 1.5x faster GET than CacheLib at 64B, 9.5x faster at 128MB working-set 2. **Vendor-independent: CacheLibNuma = CacheLib on AMD** — per-pool routing without mbind is ineffective on both platforms 3. **AMD SpinLock sensitivity amplifies lock distribution** — TL vs SN 3.4x on Milan vs 2.0x on Ice Lake 4. **CacheLib 50% memory tax confirmed on AMD** — structural, not vendor-specific 5. **4T TL anomaly resolved** — root cause: `alignas(64)` statistics atomics on heap object caused cross-socket cache-line bouncing. Fix: per-node stat sharding. 4T p50 GET: 587ns -> 71ns (-88%)

The project is on the `numa-focus` branch. Licensed under MIT. Build and test artifacts are excluded from version control.

7. Risks and Open Questions

- ~~Can lock-free reads (seqlock) reduce contention enough to expose cross-domain effects at 4+ threads?~~ **Answered: yes.** SeqLock increased the measurable cross-domain p50 overhead from 5.1% to 11.7%.
- ~~Can the 11.7% p50 cross-domain overhead measured on QEMU be replicated on real hardware with larger NUMA distances?~~ **Answered: partially.** Real hardware shows 4.5–7.0% p50 cross-domain overhead (smaller than QEMU's 11.7%, which was inflated by scheduling artifacts). The cross-domain signal is genuine but smaller than QEMU suggested.
- ~~**Is the topology-aware benefit vendor-specific (Intel only)?**~~ **Answered: no.** c6a.metal (AMD EPYC Milan, 4 NUMA nodes) confirms the same qualitative pattern as c6i.metal (Intel Ice Lake, 2 NUMA nodes). FurrBallTL beats CacheLib by 1.5–9.5x on AMD, consistent with 2.5–8.8x on Intel (§5.26).

- ~~**Does CacheLibNuma provide NUMA benefit on AMD?**~~ **Answered: no.** CacheLibNuma = CacheLib on AMD (1200ns vs 1220ns at 64MB, 2840ns vs 2830ns at 128MB). Per-pool routing without physical NUMA binding is ineffective on both vendors (§5.26).
- Will Phase 2's L2 page eviction (ARC eviction) introduce latency spikes during page flush?
- How does the bump allocator's fragmentation ratio evolve under real workloads?
- ~~Will the per-key `unique_ptr<SeqLock>` allocation overhead become a bottleneck at scale?~~ **Answered: eliminated in Phase 2a.** CMap stores values inline in cacheline-aligned slots with no per-key heap allocation.
- ~~**Hypothesized scaling advantage (untested):** Per-node sharding creates N independent contention domains~~ **Answered: validated for reads, bottleneck identified for writes.** Get throughput scales +72% from 4→64 threads (lock partitioning prevents cross-node read contention). Set throughput degrades -79% due to SpinLock bottleneck at 32 threads/node (§5.19). The read scaling confirms the hypothesis; the write bottleneck identifies the next optimization target.
- **Staging pages resolved the SET tail-latency bottleneck.** Pre-staging p99_set was 68–280K ns (O(P) page scan after eviction). Post-staging p99_set is 2.8–4.5K ns (O(1) bump or staging write). Remaining SET overhead vs TBB is architectural (ARC eviction policy, seqlock writes, CompactLock for key tracking). Further reduction would require lock-free ARC list operations or per-page eviction.
- ~~**4T TL anomaly on c6a.metal**~~ **RESOLVED in v1.33.0.** Root cause: `alignas(64)` statistics atomics on the FurrBall heap object caused cross-socket cache-line bouncing. Each GET touched 3 cache lines on node 0 from remote sockets (~360ns overhead on Infinity Fabric). Fix: per-node stat sharding in `PerNodeDetails`. 4T p50 GET: 587ns -> 71ns (-88%). The irony: cache-line isolation that prevents intra-socket false sharing maximizes inter-socket contention (§3.7, §5.26).
- **No data on 8+ NUMA nodes.** Production servers with 8+ nodes (AMD Genoa NPS=4, 4-socket Intel) would provide the strongest test of topology-aware placement. Expected to show larger TL vs CacheLib advantage as cross-domain probability increases.
- **Page-level eviction smothers policy differentiation.** Phase 2b benchmark data (§5.20) shows identical hit rates across ARC/REMARC/AUG-ADAPT in all workloads. The 2.6x latency gap is a data structure layout artifact (ArcList vs TempCtrl), not a policy advantage. Per-key EvictScore is inert for eviction. Benchmark redesign required before policy comparison can be meaningful. **Plan documented in §5.21.**
- **ARC's linked list is a performance bottleneck independent of policy choice.** ~~The 2.6x latency gap~~ **Validated and resolved in §5.22.** ARC-FLATLL2 (flat-linked-list + uint32_t + arena + no cached_) beats ARC-linked by +87–172% throughput with identical hit rates. The data structure gap is fully closed. ~~Next: REMARC + ARC-FLATLL hybrid (ARC p-adaptation + REMARC per-key scoring).~~ **Hybrid tested in §5.23: negative finding.** REMARC scoring and ARC p-adaptation are complementary, not substitutable. REMARC's value is in the migration axis (multi-action), not in replacing ARC's eviction heuristic.

- **ARC is a double-edged sword: smarter adaptation at higher per-operation cost.** ARC's p-adaptation between recency and frequency is more sophisticated than REMARC's 2-atom composition, potentially yielding better hit rates. But ArcList pointer-chasing makes each metadata operation 2-3x slower. The total throughput (hit rate x latency) tradeoff is workload-dependent and cannot be evaluated until per-key eviction is implemented.

8. Roadmap

Phase 1: Topology-Aware Core (Complete)

- ☒ Per-node physical block allocation with manual subdivision
- ☒ Bump allocator with padding waste tracking
- ☒ Key-based Set/Get with round-robin + thread-local routing
- ☒ Per-node sharded KeyStore (eliminates global map contention)
- ☒ Lock-free reads via SeqLock on KeyMeta
- ☒ CAS-based concurrent bump allocator
- ☒ Statistics with cache-line-aligned atomics (per-node sharding v1.33.0)
- ☒ Benchmark harness
- ☒ Thread-local routing comparison (round-robin vs `Numatic::GetCurrentNode()`)
- ☒ SeqLock isolation of cross-domain memory latency signal
- ☒ Baseline comparison (single-domain cache, concurrent throughput analysis)
- ☒ Simulated NUMA latency (compile-time flag, rdtsc busy-wait)
- ☒ Ablation study (5-step incremental, per-design-decision isolation)
- ☒ Shared-nothing variant (MPSC slot queue, per-node workers)
- ☒ Cross-VM baseline isolation methodology
- ☒ Real hardware validation (AWS c6i.metal, Intel Xeon Platinum 8375C, 2 sockets, 128 vCPUs)
- ☒ Thread scaling test (4/8/16/32/64 threads)

Phase 2: ConcurrentARC → REMARC

Phase 2a: Concurrent Key Store (Complete)

- ☒ CMap: concurrent Swiss table with SSE2 SIMD probing, per-slot seqlock, separate ctrl/slot allocations
- ☒ ConcurrentARC: CMap + ArcList + PromoteBuf + SpinLock
- ☒ ArcList: hash-keyed O(1) doubly-linked list for ARC tracking
- ☒ PromoteBuf: deferred MPSC promotions with cooperative drain
- ☒ UpdateInPlace: in-place value modification under seqlock
- ☒ `thread_local` cache in `GetCurrentNode()` (eliminates VM exit)
- ☒ Lock-free page advancement (`atomic<size_t> CurrentPage`)
- ☒ Phase 2a benchmark suite (ST, concurrent, routing, inline/cross-VM baseline, ablation)
- ☒ Per-node `rwMutex`, `KeyShard`, `KeyMetaStore` removed

Phase 2b: REMARC Unified Policy + Migration + Persistence

- ☒ KeyMeta additions: `Dirty` flag (seqlock-protected), `HotNode` (uint8_t), `TempCtrlIdx` (uint8_t)
- ☒ Page lifecycle: `PageTier` enum (HOT/COLD/EMPTY/FREEZE), `ActiveKeys`, `KeyIndex`, `TempCtrl`, `CompactLock`
- ☒ REMARC scoring in `Get()`: update S_local/S_remote on every hit, update HotNode probabilistically on remote Get()
- ☒ REMARC scanner: SSE2 batch scoring of TempCtrl to Evict/Migrate candidates, migration-first execution ordering
- ☒ Cross-node migration: insert-before-erase for M > theta_m candidates, HotNode as target
- ☒ Outcome-based adaptation: track eviction/migration reversal rates, adjust theta_e/theta_m
- ☒ Ghost-less reload protection: S_local=MAX on page reload from persistent storage
- ☒ Page recycling: HOT -> EMPTY -> COLD -> FREEZE -> EMPTY transitions, `ResetBump()`
- ☒ Policy framework validation: 24 variants evaluated in PolicyBench, compile-time template type parameter
- ☒ AUG-ADAPT self-tuning policy: feedback controller achieving 99.33% looping + ARC-equivalent Zipf/ScanRes/Temporal
- ☒ Phase 2b benchmark data collected on c6i.metal (§5.20: DEFERRED, page-level eviction smothers policy differentiation)
- ☒ ARC-FLATLL flat-linked-list implemented and validated: identical hit rates to ARC, +1.5–34% throughput (§5.22)
- ☒ 3-way ARC comparison complete: ARC-linked vs ARC-FLAT vs ARC-FLATLL (§5.22)

Phase 2c: Per-Key Eviction + Staging Pages

- ☒ ARC-REMARC Hybrid tested: negative finding, scoring and p-adaptation are complementary (§5.23)
- ☒ Per-key eviction: EvictOneKey removes key from Page + KeyStore, dead-bytes tracking per page
- ☒ Page recycling when ActiveKeys == 0 (Recycle + ResetBump)
- ☒ Staging pages: O(1) SET overflow target for HasStoreEviction policies (ARC, LRU)
- ☒ Page drain compaction: proactive sparse page consolidation during maintenance
- ☒ Destructor race fix: refcount + double-check replaces heuristic sleep
- ☒ CMap h2 truncation bug fix: Page stores full HashPair, not truncated uint32_t
- ☒ EC2 validation: p99_set improved 24–63x with zero GET regression (§5.25)
- ☒ ARC-FLATLL in production (CMap.h): replace std::list with flat-linked-list
- ☒ ARC-FLATLL2 optimized: uint32_t indices, pre-allocated arena, no cached_ set (+87–172% vs ARC)
- ☐ 2x2 factorial benchmark: ARC-1L vs REMARC-1L (scoring quality), REMARC-1L vs REMARC-2L (architecture benefit)
- ☐ Re-run Phase 2b workloads with per-key eviction, expect hit rate divergence

☒ Multi-platform validation (AMD EPYC Milan c6a.metal, §5.26)

Phase 2d: Two-Level REMARC + Optimization

☐ PagePolicy as explicit REMARC instance (page-level atoms/projections)
☐ Page eviction: E_page aggregation from per-key scores, page flush to RocksDB for E_page > theta_e
☐ Cold page eviction to whole-page RocksDB persistence (PageBlob format)
☐ Ghost hit recovery to whole-page RocksDB reload (speculative, full page)
☐ `evictedKeyIndex_` (H2 to PageId) for ghost hit lookup
☐ SIMD free-list optimization: uint32_t offsets, AVX2 first-fit scan, deferred (infrastructure, not policy)
☐ Multi-node simulation validation in PolicyBench (desire encoding, 2-node AUG-ADAPT)

Phase 3: Adaptive Memory Pooling (AMP)

☐ Dynamic page pool expansion under eviction pressure
☐ Page pool contraction when idle
☐ Multi-physical-block support per node

Phase 4: Dispatch Models & Optimization

4.1 Broadcast-Race Get (Revised Design)

The broadcast-race Get replaces the O(N) sequential shard scan with a parallel scatter-gather: all N node workers probe their local shard concurrently, achieving O(N) aggregate work with O(1)-round parallel latency (wall-clock time of a single shard lookup, independent of node count under fixed N workers).

Mechanism:

1. **Caller creates a result slot** initialized to a sentinel value (e.g., `nullptr`).
2. **Caller dispatches a lookup job to all N node workers** (including its own node). Dispatching to the caller's own node decouples the caller from the Furrball entirely—the caller is freed to batch additional Gets or perform other work while waiting.
3. **Each worker performs `KeyStore::find(key)`** on its local shard:
   - **Hit:** Worker writes the result data to the result slot, signals completion, and returns to its work loop.
   - **Miss (key not found or end of shard):** Worker signals failure and immediately returns to its work loop—no result write, no retry, no fallback.
4. **Caller blocks on `WaitGroup::wait()`**, then inspects the result slot: non-sentinel = hit, sentinel = total miss.

Structural single-writer guarantee: Since each key is placed on exactly one NUMA node at `Set()` time, at most one worker in the race can produce a hit. This eliminates the need for CAS, mutex, or any write-side synchronization on the result slot. There is no contention because structurally, there cannot be concurrent writes to the result.

Synchronization primitives: The only coordination is a `WaitGroup` with count N—one `Done()` per worker completion. No atomic flags, no bitmask, no polling loop. The result slot requires no lock by the structural guarantee.

Caller decoupling: By dispatching to its own node's worker (rather than performing a direct local lookup), the caller thread is temporarily decoupled from the data path until `WaitGroup::wait()` returns. This provides natural scaffolding for future batching: the caller could queue multiple Gets before waiting, amortizing the WaitGroup overhead across multiple concurrent lookups.

Waste model:

$$\text{net\_gain} = \text{latency\_saved} - (N-1) \times \text{loser\_probe\_cost}$$

where `latency_saved` is the sequential probe time eliminated and `loser_probe_cost` is the per-worker cost of an unsuccessful `KeyStore::find()` (a hash lookup returning empty). Since a miss costs only a single hash lookup with no SeqLock read, no memcpy, and no result write, `loser_probe_cost` is minimal.

Fail-safe condition: Disable the race when `loser_probe_cost × (N - 1)` exceeds sequential probe latency (race wasteful at high node counts or when the location cache miss rate is high, causing frequent all-worker dispatches with only one winner).

Partial analogy: Distributed speculative reads (Amazon Dynamo's hinted handoff). However, Dynamo operates across machines over a network; Furrballs applies the same principle across NUMA nodes within a single machine, leveraging the structural guarantee that exactly one node holds the key.

4.2 Location Cache

After a broadcast-race resolves a cold-miss location, the result (key → nodeID) is cached. Subsequent Gets go directly to the correct node's worker—no race needed. Minimal metadata (one integer per key). Eviction follows a score-based policy with tunable weights (recency-dominant by default, adjustable for empirical testing). This is the lightest form of cross-node key synchronization—cache what you found, evict what you don't use.

Consistency note: If Phase 2's ARC evicts a page and the key moves to a different node (e.g., via L2 page reload), the location cache entry becomes stale. A version counter or TTL per entry may be needed to detect stale mappings.

4.3 Remaining Phase 4 Items

- ☐ Score-based eviction policy for the location-store metadata (key → owner-node hints, including optional negative hints). This policy applies only to ownership hints used to reduce discovery/refetch cost; it does not replace the planned two-level ARC for key/page data eviction (Phase 2). Configurable weight vector (recency, frequency, cross-node access count) enables empirical testing across a range of combinations. On broadcast-race hit: cache positive owner hint. On broadcast-race miss: optionally cache short-TTL negative hint. Stale hint: rerun discovery and refresh.
- ☐ Ablation of dispatch models: sequential probe vs broadcast-race vs broadcast-race + location cache vs MPSC queue, measuring compound effect with/without NUMA pinning
- ☒ O(1) ARC lookup (hash-keyed ArcList with parallel unordered_map index, §2.8.1)
- ☒ Concurrent hash map for per-node shards (CMap, §2.7 — safe lock-free reads during writes)
- ☐ Reference counting on pages (dangling pointer safety for Phase 2b eviction)
- ☒ Theoretical scaling validation: per-node contention domains scale as `threads_per_node` rather than `total_threads`, producing smoother scaling than non-NUMA caches as thread count increases. Validated on c6i.metal (§5.19): Get +72% from 4→64 threads; Set degrades -79% (SpinLock bottleneck). Baseline single-lock Set collapses at 4 threads and stays flat.

4.4 Shared-Nothing + Broadcast-Race (Third Variant)

The SharedNothingCache's MPSC queue was an inappropriate communication model: it assumes multiple concurrent producers contending for slots (CAS-based claiming, 4-phase lifecycle), but the structural single-writer guarantee means no concurrent multi-producer access ever occurs for a given key. The MPSC overhead is pure waste.

Applying broadcast-race to the shared-nothing model eliminates the queue entirely:

Aspect	MPSC Queue (current)	Broadcast-Race
Cross-node dispatch	Caller targets specific worker via queue	Caller dispatches to all workers via WaitGroup
Slot claiming	CAS-based (unnecessary)	None (no slots)
Worker protocol	4-phase lifecycle + polling	Wake, check shard, Done()
Result synchronization	Spin-wait on `COMPLETED` atomic	WaitGroup + single-writer result slot (no CAS)
Overhead source	Queue protocol (~1,500ns)	N-1 unsuccessful hash lookups (~100ns each)

The resulting model is a third variant combining shared-nothing data access with broadcast-race discovery. Workers still access only local memory (shared-nothing property preserved), but key discovery uses parallel probing instead of point-to-point queue communication. Projected cross-node latency: ~1 shard lookup + WaitGroup dispatch overhead, vs ~1,500ns for the MPSC queue.

- ☒ Implement shared-nothing + broadcast-race variant (implemented, evaluated in QEMU, deferred: CV wake overhead ~24us in QEMU masks any real cross-domain effect; requires real hardware to validate)
- ☐ Three-way ablation on real hardware: MPSC queue vs broadcast-race vs shared-nothing + broadcast-race

Phase 5: Publication

- ☐ Formal paper with reproducible benchmarks
- ☐ arXiv submission under `cs.DC`
- ☐ PhD application material

9. Conclusion

Phase 1 of Furrballs demonstrates that asymmetric memory topology (NUMA being the most widespread instance) can serve as a meaningful input to cache placement decisions. The architecture—per-node physical allocation, bump-packed multi-value pages, and ARC eviction—provides a structured foundation for topology-aware caching where the **page** is the locality unit and the **key** is the access unit.

The critical Phase 1 finding is that synchronization overhead masks the true cross-domain memory latency signal: under `shared_mutex`, cross-domain overhead was 5.1% at p50; under SeqLock lock-free reads, it is 11.7%—more than double. Thread-local routing with SeqLock achieves 26–41% improvement over round-robin, confirming that topology-aware placement provides compound benefits. Per-domain sharding provides 3x concurrent Set throughput through natural lock partitioning at a mixed single-threaded cost (Set +3%, Get +17%).

Phase 2a replaces the Phase 1 data structures with a concurrent Swiss table (CMap) and ARC policy (ConcurrentARC). The results reveal a tradeoff: single-threaded latency increases by 6–41% (architectural cost of CAS-based concurrency and ARC list management), but concurrent Set throughput improves by 58% (per-slot CAS eliminates shared_mutex serialization). CMap resolves the Phase 1 data race on `unordered_map::find()` during concurrent reads, making the system safe under mixed read/write workloads. PromoteBuf recovers concurrent Get from 0.56x to 0.80x vs baseline by deferring ARC promotions out of the critical read path. Hash-keyed ArcList eliminates all string copies from ARC tracking, and a `thread_local` cache in `GetCurrentNode()` eliminates ~1000ns VM exit per routing decision.

Phase 2b initially implemented REMARC (Reduction-Modeled Adaptive Replacement Cache) as a unified eviction/migration policy, but extensive evaluation proved that the REMARC framework adds zero value over simple baselines. The companion REMARC paper [Sphynx2025] (v1.0.0, DOI: 10.5281/zenodo.19794758) documents the full evaluation: score degeneracy (§10), tiered placement simulation across K=2 and K=3 nodes with 10+ strategies (§11), and theoretical analysis of feedback signal quality (§11.5). Key findings: (1) the REMARC formula contributes zero

eviction improvement over LRU, (2) all REMARC migration variants are equivalent to simple EMA counting, (3) feedback for placement decisions has SNR 0 (migration success rate ~50% regardless of strategy), and (4) "do nothing" (static placement) beats all adaptive strategies on half the tested workloads. The production path now uses ARC for eviction (proven, well-understood) and SimpleMigratePolicy for migration (two 4-bit EMA counters per key, no lookup tables, no SIMD).

The five-step ablation study (§5.10, §5.17) isolates the contribution of each architectural decision across both phases. The Phase 2a ablation reveals that the `thread_local` cache fundamentally changed the routing dynamics: thread-local routing is now *faster* than round-robin (ablation D < C), unlike Phase 1 where the `GetCurrentNode()` syscall made it 34% slower. This has a methodological implication: Phase 1's routing overhead findings (26–41% improvement from thread-local routing, ablation C→D +34% Set increase) are **environment-specific** rather than architectural. They reflect QEMU's VM exit cost on `sched_getcpu()`, not an inherent property of topology-aware routing. On real hardware where `GetCurrentNode()` costs ~1–10ns via VDSO, the Phase 1 routing story would resemble Phase 2a's.

Real hardware validation on AWS c6i.metal (Intel Xeon Platinum 8375C, 2 sockets, 128 vCPUs) confirms the architecture's NUMA benefit: a genuine 4.5–7.0% p50 cross-node overhead (§5.18), thread-local routing providing +60–65% improvement over round-robin, and ablation step D (thread-local routing) as the primary driver of the cross-node signal (+21.0–38.9% Cross-OH). Get throughput scales +72% from 4→64 threads (§5.19), validating the lock partitioning thesis for reads. Set throughput degrades -79% at 64 threads due to per-node SpinLock contention, identifying striped SpinLocks or per-bucket ARC locks as the next optimization target. The QEMU results validated the methodology and architecture; the real hardware results validate the performance claims.

Cross-platform validation on AWS c6a.metal (AMD EPYC 7R13 Milan, 4 NUMA nodes, 192 vCPUs) confirms the topology-aware benefit is vendor-independent (§5.26): FurrBallTL achieves 1.5x lower GET latency than CacheLib at 64B and 9.5x lower at 128MB working-set (UniformRO 1024B). The advantage persists across two vendors (Intel/AMD), two NUMA topologies (2-node/4-node), and two interconnects (UPI/Infinity Fabric). CacheLib's per-pool NUMA routing (CacheLibNuma) is ineffective on both platforms, confirming that CacheLib has no working NUMA-aware memory placement in any open-source release. The working-set sweep is the definitive comparison: at capacities exceeding L3, DRAM access reveals the full topology-aware advantage, with FurrBallTL at 128MB achieving 300ns p50 GET vs CacheLib's 2840ns (9.5x) at half the memory footprint.

The system's engineering contributions remain sound: topology-aware allocation, concurrent ARC, bump-packed pages, and per-domain sharding are independent of the REMARC policy framework. The negative findings on REMARC (cold-entry information hole, feedback SNR 0, feedforward expressiveness boundary) are documented in the companion paper and contribute to the community's understanding of the limits of adaptive resource management.

Appendix A: Repository Structure

```
Furrballs/
  Furrballs/          # Core library (C++20, MIT license)
    include/
      Furrballs.h     # Public API, ARC, configs, KeyMeta
      CMap.h          # Concurrent Swiss table (CMap), ArcList, PromoteBuf, ConcurrentARC, Spi
      Page.h          # Page struct with bump allocator, TempCtrl, KeyIndex, HotNodes
      Policy.h        # SimpleMigratePolicy (production), ArcPolicy, RemarcPolicy, StandardRema
      Remarc.h        # REMARC lookup tables, RemarcConfig, SIMD scanner
      Logger.h        # Thread-safe logging
      Error.h         # Categorized error enum
      NodeJob.h       # Per-node worker threading
      WaitGroup.h     # StreamLine synchronization (copied)
      Concept.h       # StreamLine Lockable concept (copied)
      SeqLock.h       # StreamLine SeqLock for lock-free reads (copied, used by BaselineCache)
      MemoryManager.h # General allocation, padded_size utilities
    src/
      Furrballs.cpp   # Core: Bootstrap, CreateBall, Set, Get, Flush, Evict
      Logger.cpp      # Platform-safe time formatting
      NodeJob.cpp     # Worker loop and condition-variable dispatch
      Page.cpp        # (Empty; Page is header-only)
  FurrballsSM/        # Shared-nothing variant (MPSC queue, per-node workers)
    include/
      SharedNothingCache.h  # Public API (same Set/Get interface)
    src/
      SharedNothingCache.cpp # MPSC slot queue + worker implementation
  numatic/            # NUMA platform abstraction
    include/Numatic.h  # Free function declarations + CRTP PMR allocators
    src/
      NumaticUnix.cpp  # Linux implementation (libnuma, thread_local GetCurrentNode cache)
      NumaticWin.cpp   # Windows stubs/heuristics (thread_local GetCurrentNode cache)
  Benchmark/          # Benchmark harness
    BaselineCache.h   # Non-NUMA control group (malloc, single map, SeqLock)
    NUMAAllocCache.h  # Ablation step B (per-node alloc, single map, round-robin)
    Benchmark.cpp     # Full benchmark suite (runs in NUMA VM)
    BaselineBenchmark.cpp  # Standalone baseline benchmark (runs in non-NUMA VM)
  Sandbox/            # Unit tests
  docs/               # This whitepaper
```

Appendix B: Build and Run

```
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --target Benchmark
./Benchmark/Benchmark
```

QEMU VM:

```
~/vm/furrballs/furr.sh benchmark  # Build on host, run in VM
```

Appendix C: Changelog

Version	Summary
v1.32.0	Cross-platform NUMA validation on c6a.metal (AMD EPYC Milan, 4 NUMA nodes). FurrBallTL 1.5x faster GET than CacheLib at 64B, 9.5x at 128MB. CacheLibNuma = CacheLib on AMD. 212 benchmarks, 12 adapters, zero crashes. §5.26 multi-platform results, §9 conclusion updated.
v1.31.0	Staging pages: O(1) SET overflow for ARC/LRU policies, 24–63x p99_set improvement on c6i.metal. Page drain compaction. Destructor race fix (refcount + double-check). CMap h2 truncation fix (full HashPair storage). §2.10 staging design, §5.25 EC2 results.
v1.30.0	ReadOnly key prefix bug fixed. CacheLib v2026.02 validated (no perf change vs v2024.07). NUMA DRAM penalty: 171ns (64MB), 211ns (128MB). CacheLib Memory Tiers not implemented. FurrBallSN 10.9x/13.9x faster GET than CacheLib at equal capacity.
v1.29.0	Corrected-capacity NUMA comparison at 1024B. Remote-Only data: 43ns = SN 43ns, proving 32MB fits in L3. TL-vs-CN gap is cache locality not DRAM locality. Memory efficiency: FurrBall 100% vs CacheLib 50%. CN 191ns = cross-socket L3 metadata traffic.

Version	Summary
v1.28.0	Preliminary multi-baseline NUMA comparison (superseded by v1.29.0 capacity fix).
v1.25.0	Strategic closure. REMARC retired. Production path: ARC eviction + SimpleMigratePolicy. REMARC paper v1.0.0: comprehensive tiered placement evaluation (§11), feedback SNR analysis, REM-T algebra retired.
v1.24.0	REMARC paper v0.13.0 critical correction. Sentinel bug + score degeneracy invalidate HA hit rates. REMARC paper old §10-§11 removed. New §10 (post-publication audit). Whitepaper REMARC references corrected.
v1.22.0	History Atom experiment (REMARC paper old §10, Findings 35–38; retracted v0.13.0). REMARC paper v0.11.0. PolicyBench 36 variants.
v1.20.0	§5.23 HYBRID-POS and HYBRID-FB results. Cross-reference to REMARC paper Findings 32–34.
v1.19.0	§5.23 ARC-REMARC Hybrid: negative finding. REMARC scoring and ARC p-adaptation are complementary, not substitutable.
v1.18.0	Strategic reframe to topology-aware asymmetric memory. ~90 substitutions across all sections.
v1.17.0	§5.22 ARC-FLATLL2 limits: 15/15 pass, 8 documented limits. `--limits` mode added.
v1.16.0	§5.22 ARC-FLATLL2 optimized variant: uint32_t indices, pre-allocated arena. +87–172% over ARC.
v1.15.0	§5.22 ARC-Flat data structure improvement. ARC-FLATLL +1.5–34% over ARC-linked.
v1.14.0	§5.21 per-key eviction design plan. Two-level REMARC architecture. Roadmap Phase 2c/2d.
v1.13.0	Phase 2b benchmark on c6i.metal. §5.20 DEFERRED: page-level eviction smothers policy differentiation.

v1.12.0	REMARC paper v0.8.0 Finding 31. Consistency pass: 53 issues fixed across both papers.