NICMM: A Memory Management Runtime for On-Path SmartNICs

Under submission - Please do not distribute

Kshitij Rana Landon Johnson Xiao Zhang Daehyeok Kim

The University of Texas at Austin

Abstract

On-path SmartNICs expose heterogeneous memories—small, fast on-chip memories and larger, slower off-chip memory. Developers typically reserve state capacity and choose memory tiers at compile time. Static allocation and placement strand capacity and leave frequently accessed state in the slow tier as demands change.

We present NICMM, a runtime that dynamically allocates and places NIC state while sharing fast-tier capacity across applications. Our key insight is to let the runtime resolve applications' state keys, so it can move objects without applications needing to track where those objects reside in memory. NICMM exposes a key-value API that makes allocation, placement, and sharing transparent to applications, freeing developers from managing objects across memory tiers.

We evaluate our Netronome Agilio NIC-based prototype across four network functions with flow churn and skewed accesses. NICMM reduces median packet processing latency by 29–40% relative to static slow-tier placement while admitting state beyond fast-tier capacity. It also lowers tail latency, delivers comparable throughput with modest memory overhead, and reclaims borrowed capacity within seconds.

1 Introduction

On-path SmartNICs execute stateful applications directly on the packet-processing path [11, 15]. They allocate and access *state objects*, such as flow entries, address mappings, and counters. The bare-metal NICs we target expose small, fast on-chip memories and larger, slower off-chip memory, and leave developers to reserve capacity and assign objects to these memory *tiers*, typically at compile time [1, 2, 6, 7]. Underprovisioning a data structure can reject new state despite idle capacity in other data structures, while overprovisioning wastes reserved capacity. Moreover, placing frequently accessed objects in slower tiers increases lookup latency. Static provisioning therefore forces developers to trade off memory capacity against access latency.

Keeping frequently accessed, or *hot*, objects in fast tiers can reconcile latency and capacity, but only if placement tracks changing demand. Our data-center trace analysis shows that most packets access a small hot set, but the objects comprising this set change over time (§2.2.2). Further, when several applications share a NIC, their capacity demands also change independently. Static fast-tier partitions can leave capacity unused while another application needs it, whereas unrestricted sharing lets one application dominate.

Memory managers designed for CPUs suggest two approaches, each addressing part of this problem. An OS centrally manages memory and moves pages, fixed-size blocks, between memory tiers [22, 25]. Such *page-based tiering* can move cold objects with hot ones, wasting fast-tier capacity [10]. Alternatively, application-specific managers could move individual objects, following record-level movement in database and storage indexes [26, 29]. These managers exploit local access patterns but leave developers to coordinate memory capacity across applications. Hence, NICs need object-level decisions with coordinated memory sharing.

We present NICMM, a runtime for dynamic object allocation, placement, and fast-tier sharing across NIC applications. Our key insight is that applications already identify state by keys, such as a flow's five-tuple, independently of physical location. A *key-value API* lets applications create, read, update, and delete state by key, freeing developers from managing allocation, reclamation, and movement across memory tiers. The runtime allocates new objects from each application's slow-tier partition and reuses freed memory. It *promotes* hot objects to faster tiers and *demotes* colder objects to slower tiers. We separate placement and sharing policies from allocation, state access, and movement, letting operators configure policies without modifying applications.

Realizing this abstraction on the NIC presents challenges because management overhead can offset the benefits of fast-tier access. Dynamic allocation must accommodate hundreds of workers without excessive lock contention or metadata that crowds out application state. Once allocated, hot objects benefit from promotion only if lookups can reach them without traversing slow-tier objects or waiting for movement to finish. With multiple applications, these benefits also depend on when borrowed capacity is reclaimed: returning it too early displaces useful hot state, while returning it too late keeps other applications' hot objects in the slow tier.

We address these challenges through four key insights. First, to support growing and shrinking state with low contention and metadata overhead, we devise a *hierarchical size-class allocator*. It builds on two observations: size classes can coordinate independently, and a bulk reservation serves many allocations. Shared size-class pools reuse freed blocks, while

bulk provisioning amortizes coordination without per-worker free-object caches or replicated metadata.

Second, to keep hot-object lookups fast as state grows, we propose a *hotness-ordered hash table*. Our insight is that adding cold state need not change how hot objects are located. We locate fast-tier objects by hashing into fixed arrays, regardless of how much state resides in the slow tier.

Third, to keep lookups fast during updates and movement, we develop *read-optimistic concurrency* across memory tiers. Inspired by optimistic concurrency control [21], we exploit the insight that lookups need not prevent concurrent changes if they can detect interference and safely retry. Version validation checks fast-tier reads, while deferred storage reuse protects ongoing lookups without locks.

Finally, to balance fast-tier borrowing with application shares, we propose *demand-triggered sharing*. We observe that a failed promotion reveals competing demand when a hot object needs a candidate slot occupied by another application. Reclaiming borrowed capacity only under competing demand preserves work-conserving sharing.

We implement NICMM as a Micro-C runtime linked into application firmware on the Netronome Agilio NIC. With its API, we develop four network functions: a firewall, NAT with flow metering, an L4 load balancer, and virtual private cloud (VPC) forwarding. We will open-source the implementation.

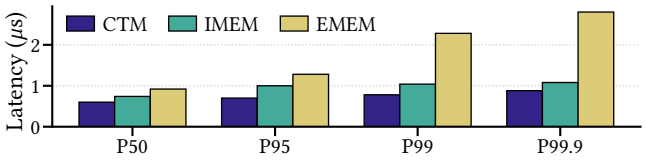
We evaluate NICMM on a Netronome Agilio CX 40GbE testbed using these applications. With flow churn and skewed accesses, NICMM serves hot entries from fast tiers while admitting working sets beyond fast-tier capacity. This reduces median packet processing latency by 29–40% relative to static slow-tier placement and also lowers tail latency. Co-located applications borrow idle fast-tier capacity and return it within seconds of competing demand. NICMM maintains comparable throughput with modest memory overhead.

2 Background and Motivation

2.1 Primer on On-path SmartNICs

This paper targets *on-path* SmartNICs, whose programmable cores inspect and process every packet directly on the data path. Examples include the Netronome Agilio [1], Intel IPU [6], AMD Alveo [2], and AMD Pensando Elba [7]. To sustain high throughput (*e.g.,* 40–200 Gbps), these devices employ a massively parallel architecture with many concurrent *workers*. Workers are hardware threads on network processing unit (NPU)-based NICs (*e.g.,* Agilio) or pipeline instances on FPGAs (*e.g.,* Alveo). They also expose multiple memory resources ranging from small, fast on-chip SRAM to large, slower off-chip DRAM or HBM. These data planes typically run bare-metal firmware or bitstreams with direct access to physical memory, without a hardware memory management unit (MMU).

Target platform. We chose the Netronome Agilio NIC [1] for its commercial availability and Micro-C SDK. Its NFP-4000



(a) Lookup latency in each memory tier.

Tier	CTM	IMEM	EMEM
Max. flows	32 768	65 536	67 108 864

(b) Capacity of each memory tier (maximum flows).

Figure 1: Static placement's latency-capacity tradeoff for the firewall application.

architecture (§A) features 60 flow processing cores (FPCs). Each supports 8 hardware threads, yielding 480 concurrent workers. Its memory hierarchy contains three tiers with different capacities and access latencies. Cluster Target Memory (CTM) provides 256 KiB per island (a group of FPCs), Internal Memory (IMEM) provides 4 MiB, and External Memory (EMEM) provides 2 GiB. We call on-chip SRAM (CTM and IMEM) the *fast tiers* and off-chip DRAM (EMEM) the *slow tier*. We discuss portability to other NIC architectures in §10.

Stateful NIC programs. On-path NIC programs maintain data structures such as connection-tracking tables, NAT tables, and metering counters. An individual entry, such as one flow's NAT entry, is an *object*: the unit of state allocation and access. Its memory tier affects performance: fast SRAM reduces access latency, while slower DRAM accommodates large working sets. Yet these SmartNICs offer little runtime support for *dynamic memory allocation* or *data movement* across tiers. Developers must reserve each data structure's capacity and assign it to a memory tier, typically at compile time.

2.2 Limits of Static Allocation and Placement

Static memory management fixes three decisions before demand is known: per-structure capacity, object placement, and capacity shares across applications.

Static sizing requires predicting peak object populations. Underprovisioning rejects new state even when memory reserved for other structures in the same application is idle, while overprovisioning strands capacity. This mismatch persists even when all structures reside in the larger slow tier: a table that fills its reservation cannot use another table's spare space. Dynamic allocation lets structures acquire storage for new objects and release it after deletion. Where that storage resides creates a second problem: each tier trades capacity for access latency.

2.2.1 Latency-Capacity Tradeoff

We use a connection-tracking firewall that maintains 20-byte flow objects and accesses its state table once per packet. We compare three compile-time configurations that place the complete table in CTM, IMEM, or EMEM. To isolate tier latency, all configurations use the same 32,768 objects and packet trace and run below saturation. We measure capacity

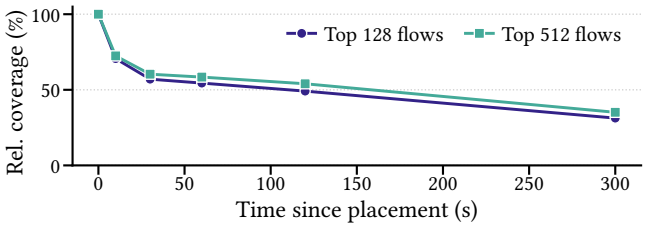


Figure 2: Packet coverage of frozen top-K objects, normalized to each window's top-K oracle (median).

separately by increasing active flows until insertion fails, including table metadata in the footprint. Figure 1a shows that CTM reduces P50 and P99.9 lookup latency by 34.8% and 68.6% relative to EMEM. However, it holds three orders of magnitude fewer flows (32 K *vs.* 64 M; Figure 1b). IMEM falls between these endpoints. Placing the complete table in one tier therefore sacrifices either capacity or access latency.

2.2.2 Time-Varying Object Demand

Placing only frequently accessed *hot* objects in the fast tier could reconcile latency and capacity if accesses concentrate on a small set of objects. A static placement further requires that set to remain stable. We test both conditions using data-center packet traces containing 116.1 million TCP and UDP packets [12]. Each bidirectional five-tuple represents one fixed-size flow object. We divide the traces into non-overlapping 10-second windows and exclude empty intervals between capture segments, leaving 1,262 windows. An object's packet count within a window defines its hotness.

The hottest 128 objects per window account for a median 95.8% of packets, despite a median of 821 active objects per window. A small fast tier can therefore cover most accesses.

To test stability, we freeze each window's top-K objects for $K \in \{128, 512\}$ and measure the fraction of packets accessing them in later windows. We normalize this coverage by an oracle that selects the evaluated window's top-K objects, giving the maximum coverage possible with K objects. We measure elapsed time between window starts and report the median across window pairs, excluding pairs that cross capture gaps. A frozen 128-object selection achieves only 70.6% of the oracle's coverage 10 seconds later and 31.3% 300 seconds later (Figure 2). Static placement therefore fails to keep hot objects in the fast tier as demand changes.

2.2.3 Multi-Tenant Memory Sharing

An operator may also colocate applications from multiple tenants on one NIC. Two approaches are to reserve a separate fast-tier region for each application's state or let tenants share a data structure. Each sacrifices either utilization or capacity isolation. We evaluate these tradeoffs in §9.3.

Static partitions underutilize capacity. The operator can reserve a separate fast-tier region for each application's state, such as a connection-tracking table. Suppose two applications with equal-size objects each receive half of a fast tier, but demand 80% and 10% of its capacity, respectively. The layout leaves 40% of the tier idle, yet the first application's unmet demand is 30% of tier capacity. Partitions protect capacity shares but prevent borrowing across applications.

Shared tables lack tenant isolation. Tenants running the same application can instead store their state in one table spanning the tier, with tenant identifiers included in keys. This approach is work-conserving: any tenant can use otherwise idle capacity. Without per-tenant capacity limits, however, one tenant can occupy most entries and crowd out others. It also cannot combine different data structures.

These limitations call for dynamic allocation and freeing, adaptive object placement, and coordinated fast-tier sharing.

2.3 NIC Memory Management Design Space

Memory management must choose a *granularity* for allocation and movement and a *scope* for coordinating capacity. Two approaches from CPU host systems offer partial solutions.

Option 1: Coarse, fixed-size block management. Conventional OS tiering allocates and moves pages [22, 25], which work well when nearby objects share access patterns. NIC objects span tens to hundreds of bytes, but interleaved flow arrivals mix hot and cold objects within pages, as in host memory tiering [10]. Moving a 1 KiB page for one hot 64 B object transfers 16× the required data and consumes 16× the required fast-tier capacity.

Page-based systems hide movement by translating stable application-visible addresses to current physical locations. Mapping the NFP's 2 GiB EMEM with 1 KiB pages and 64-bit entries requires a 16 MiB single-level page table, exceeding combined on-chip capacity. An uncached off-chip translation adds a DRAM access even for fast-tier objects.

Option 2: Per-data-structure management. Similar to record-granular movement in database and storage indexes [26, 29], each NIC data structure could allocate and move individual objects, avoiding page-level waste. However, independent managers see only their own allocations, while fast-tier capacity is shared across structures and applications. Without coordination, partitions strand idle capacity while an unenforced pool lets one application dominate (§2.2.3).

These limitations motivate a common NIC runtime for object allocation, access, and movement, with fast-tier capacity coordinated across applications. Placement policies choose objects' tiers, while sharing policies arbitrate fast-tier capacity across applications. Keeping these decisions separate from runtime operations lets operators adapt to workloads without reimplementing memory management for each application.

3 NICMM Overview

We present NICMM, a memory management runtime for on-path SmartNICs. NICMM allocates and moves state objects (*e.g.,* flow entries, NAT bindings, or counters) while coordinating fast-tier capacity across applications.

3.1 Design Goals

We design NICMM around three goals.

G1: Minimal runtime resource and performance overhead. Runtime metadata should consume minimal on-chip memory. Also, per-memory-access overhead must remain small enough to preserve the latency benefit of fast-tier placement.

G2: Work-conserving fast-tier sharing. Applications should have target shares of fast-tier capacity under contention, configured by the operator through application weights. When an application does not need its share, others should be able to use that idle capacity, making sharing work-conserving.

G3: Transparent memory management. The runtime should reduce developer burden by handling allocation, placement, and movement, so applications need not implement these mechanisms, fix per-structure capacities, or synchronize with relocation. Application logic should remain independent of tier count and capacity. Operators should be able to configure memory policies without modifying applications.

3.2 Our Approach

NIC state is naturally identified by keys: a firewall, for example, retrieves connection state using a flow's five-tuple. We exploit this property to place memory management behind a *key-value API* for insertion, lookup, update, and deletion. Applications supply stable keys; NICMM resolves each access, so applications need not track objects' physical locations.

We develop a *tiering-based design* that uses the tiers' combined capacity without requiring slow-tier backing copies for fast-tier objects. New state is allocated dynamically from each application's slow-tier partition, whose capacity is fixed at initialization. Deletion releases storage for reuse, so data structures can grow and shrink within the partition without fixed per-structure reservations. As demand changes, the runtime *promotes* hot objects to faster tiers and *demotes* colder objects to slower tiers. The runtime coordinates placement across applications to keep hot objects in fast tiers.

Different applications may favor different tradeoffs between access latency and capacity sharing. Our *configurable policy interface* separates placement and fast-tier sharing decisions from allocation, state access, and safe movement, letting operators configure policies without modifying applications. These mechanisms could also support caching (§10).

3.3 Challenges

Realizing this approach while satisfying the design goals presents four challenges.

C1: Scalable object-granular allocation. Growing and shrinking state requires hundreds of workers to allocate and recycle memory within each application's slow-tier partition. A single allocator lock serializes workers. Per-worker caches reduce contention, but each worker holds its own free blocks and tracking metadata. Both contention and on-chip metadata must remain small.

C2: Fast-tier lookup without slow-tier accesses. Moving a hot object to a fast tier does not necessarily make it fast to find. In a chained hash table, finding an object may require following links through other objects stored in slower tiers. The challenge is to let the table grow without putting those slow-tier accesses on the path to hot objects.

C3: Safe movement with low lookup overhead. Workers continue accessing objects while the runtime moves them. A lookup can see a partial value, miss an object, or reach storage reused for another object. Because lookups lie on the per-packet path, preventing these errors without per-lookup locking is critical to performance.

C4: Returning borrowed fast-tier capacity. One application's unused fast-tier share could accelerate another application's hot objects. Applications should be able to borrow it beyond their configured shares, then yield it when other applications need it. Early return demotes useful state; late return keeps other applications' hot objects in the slow tier.

3.4 Key Ideas

We address these challenges with four key ideas.

I1: Hierarchical size-class allocator (§4). To support growing and shrinking state efficiently, we combine shared size-class pools with bulk provisioning. The insight is that requests for different size classes can coordinate independently, while one large reservation serves many allocations. Shared pools avoid separate free-object caches and metadata per worker.

I2: Hotness-ordered hash table (§5). To keep hot-object lookups fast as state grows, we separate fixed fast-tier slots from a dynamically allocated slow-tier table. The insight is that growing cold state need not change where a lookup searches for hot objects. Hashing computes fast-tier candidates directly; the base lookup reaches the slow tier only when no fast-tier candidate matches.

I3: Read-optimistic concurrency (§6). To preserve lookup speed during updates and movement, we develop *read-optimistic concurrency* across tiers. The insight from optimistic concurrency control [21] is that lookups can validate their observations and retry when concurrent changes interfere. Version validation covers fast-tier reads and misses, while deferred storage reuse protects ongoing lookups without locks. Writers still synchronize.

I4: Demand-triggered sharing (§7). To return borrowed fast-tier capacity under competing demand, we base reclamation on unsuccessful promotions. A promotion blocked by another application's object exposes both the demanded slot and its owner. Our default policy combines this signal with excess occupancy and hotness to select background demotions, preserving borrowing when others do not need their shares.

3.5 System Architecture

Figure 3 shows how NICMM combines these building blocks in its tiering design. Applications register data structures through NICMM's APIs (§B; §E). Operators configure slow-

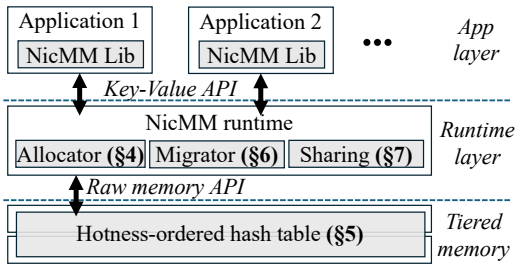


Figure 3: NICMM architecture overview.

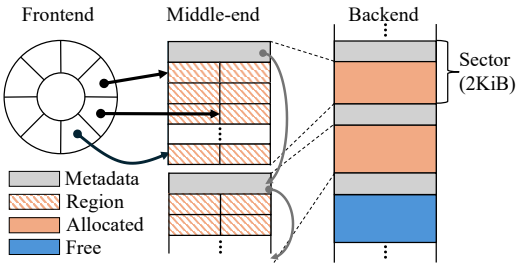


Figure 4: Hierarchical allocator: shared block reuse, size-class tracking, and sector provisioning.

tier partition sizes, fast-tier sharing weights, and placement thresholds.

Packet-processing workers. Workers use NICMM's key-value API to insert, look up, update, or delete application state. After packet egress, the NICMM runtime records the access and may promote the object on the same worker.

Background maintenance. The NICMM runtime runs one *migrator* thread per group of workers sharing a local fast tier. The runtime periodically decays access counts and selects objects for demotion based on these counts, recorded demand, and configured application shares. It also reclaims removed slow-tier objects on this thread once ongoing lookups can no longer access them, returning their memory to the allocator.

4 Scalable Dynamic Memory Allocation

Data structures must acquire storage when objects are created and release it after deletion. The challenge is to coordinate hundreds of workers accessing shared free memory without serializing their requests or consuming scarce on-chip capacity with allocation metadata.

Strawman solution. A single allocator lock serializes workers even when their requests concern unrelated objects. CPU allocators such as TCMalloc [8] reduce contention with per-thread or per-CPU caches of free objects. These caches require each worker to hold free objects and tracking metadata, consuming on-chip memory needed for application state.

4.1 Limiting Contention in Two Dimensions

NICMM combines size-class partitioning with hierarchical allocation to limit contention scope and coordination frequency.

Size-class partitioning limits contention scope. Requests mapped to different size classes need not coordinate over the same free pool. NICMM rounds each request up to a supported *size class* and returns a *block* of that size from a pool shared across workers. This separates contention by size without keeping private pools per worker. Small state objects in typical NIC applications motivate 8 B increments through 72 B, with larger classes up to 1 KiB.

Hierarchical allocation reduces coordination frequency. Obtaining a larger region once serves many allocations; reusing freed blocks avoids revisiting that region's metadata. Like TCMalloc [8], our hierarchy separates reuse from bulk provisioning, but shares frontend pools across workers (Figure 4). The *frontend* reuses blocks through rings; the *middle-end* divides regions called *sectors* into blocks; the shared *backend* supplies and recycles sectors.

4.2 Frontend: Lockless Per-Size-Class Pool

A shared ring per size class stores reusable block locations. Agilio's NFP provides hardware atomic ring insertion and removal, avoiding software locks.

An allocation first removes a block from the ring; if the ring is empty, it obtains a block from the middle-end. A free returns the block to the ring when space is available, or to the middle-end when the ring is full. Initialization populates each ring with a configurable number of blocks; ring hits avoid middle-end synchronization. Because workers share these rings, adding workers requires no additional cached blocks or tracking metadata.

4.3 Middle-End: Per-Sector Region Tracking

The middle-end handles requests that frontend rings cannot serve. NICMM embeds an allocation bitmap, free-block count, and same-class sector links in each sector. Metadata therefore scales with the managed pool and resides in the same tier, avoiding a separate on-chip tracking table. One memory read fetches the bitmap for local free-block selection.

Each size class has lists of full and non-full sectors protected by a class-specific lock. Allocation marks a free block in a non-full sector's bitmap, requesting a new sector only when none has space. A block cached in the frontend remains marked as allocated in the bitmap until returned to the middle-end. Once all blocks in a sector have been returned, the sector becomes available to the backend for reassignment.

4.4 Backend: Sector Provisioning

Released sectors can serve any size class, keeping capacity flexible across classes. The backend provisions 2 KiB sectors by default from a lock-protected physical-memory free list and coalesces adjacent freed regions to reduce external fragmentation. Backend requests still contend, but each sector amortizes the shared lock across many object allocations.

5 Hotness-Ordered Hash Table

The allocator supplies storage across tiers, but finding hot objects may still require traversing cold state in slower tiers.

5.1 The Cross-Tier Traversal Problem

NIC applications use hash tables to map keys to dynamically allocated state. In a chained table, each entry in a *bucket array* points to a linked list of objects whose keys hash to that bucket. Following these lists under tiering can cause *cross-tier traversal*: reaching a hot object requires accessing objects in slower tiers.

Suppose only the last object in a three-object chain is promoted. A lookup still reads the bucket pointer and two slow-tier objects, costing $T_B + 2T_{\text{slow}} + T_{\text{fast}}$, where T_B is the bucket-array latency and T_{slow} and T_{fast} are the tier latencies. Promotion accelerates only the final access.

Strawman solutions. An indirection table maps object IDs to locations, but grows with object count and may require slow-tier access. Updating pointers when objects move—including the next-object pointers connecting a chain—avoids indirection but requires tracking pointers and synchronizing updates with lookups. Neither approach removes the slow-tier objects traversed to reach a hot object.

5.2 Hotness-Ordered Layout

To eliminate cross-tier traversal, we propose a *hotness-ordered hash table* that separates preallocated fast-tier arrays from a dynamic slow-tier table (Figure 5). Our traces show that a small hot set lets limited fast-tier capacity cover most accesses (§2.2.2). The key insight is that additional state can grow in the slow tier without changing how these hot objects are located. This *tier isolation* lets hashing locate a fast-tier candidate without accessing the slow tier.

Lookups first check the hash table's *head* (fast-tier arrays), then its *tail* (slow-tier table) on a miss. The head holds live state without a backing copy in the tail. Migration between tiers briefly retains both copies: the destination becomes visible before the source is removed, preserving live state during concurrent access (§6).

Head: set-associative array. The head preallocates S sets of W fixed-size slots (ways) to fit fast-tier capacity. Both S and W are powers of two. The object's set index is $i = \text{hash} \,\&\, (S-1)$; the next $\log_2 W$ hash bits select its way within that set. NICMM checks this hash-selected slot for the key and data-structure owner, so each object has one candidate per tier. An occupied candidate may block promotion; lookup never scans additional slots in that tier. Using an empty candidate requires no allocation or free-list search.

With multiple fast tiers, NICMM maintains an array per level, computing each candidate independently from the same hash. Lookup checks tiers from fastest to slowest.

Slot layout. Fixed-size slots leave unused bytes when an object's payload is smaller than the slot's capacity, but enable direct indexing and allocation-free placement. The dynamic tail instead allocates each object from its size class. The flow entries, NAT bindings, connection state, and counters we target occupy tens of bytes. Our prototype supports key-value payloads up to 56 B. Each slot also has a hash, version counter for read validation, access counter for hotness, valid flag, and owner ID (Figure 17 in §D). Owner IDs distinguish data structures sharing an array and identify applications for occupancy and weighted sharing (§7). Firewall and NAT entries remain distinguishable even when their keys and slots match.

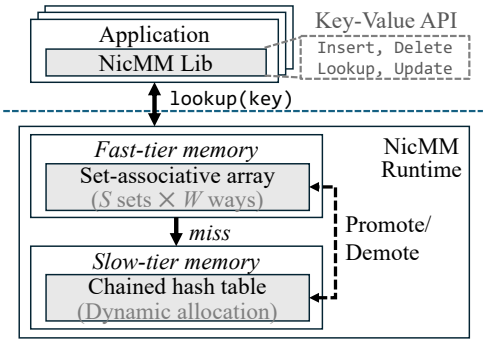


Figure 5: Hotness-ordered hash table: fixed fast-tier arrays and dynamically allocated slow-tier chains.

Tail: chained hash table. The tail uses the hierarchical allocator (§4) to grow chains of newly inserted and demoted objects within each application's slow-tier partition. Longer chains do not increase the cost of checking fast-tier slots.

Optimization with tail cache. An optional fast-tier *tail cache* speeds slow-tier access by bypassing unsuccessful fast-tier checks. It stores tail-object hashes as hints, favoring higher access counts on conflicts. A match directs lookup to the tail to verify key and owner. Missing hints or failed tail lookups use the base path. Stale hints can add a tail lookup.

5.3 Object Lifecycle

After packet egress, the same worker increments the accessed object's counter. The migrator discounts older fast-tier accesses using a configurable decay fraction ρ and scan interval T_d. The default placement policy uses these counts as follows.

Insert. New objects enter the corresponding slow-tier chain if allocation within the application's partition succeeds (§7). This avoids consuming fast-tier capacity and moving objects before they demonstrate frequent access.

Promotion. When the access count reaches tier l's promotion threshold P_l and the candidate slot is available, the worker promotes the object to the next faster tier. Promotion from the slow tier carries over the source count, capped below the destination's promotion threshold. This avoids immediate demotion but requires more accesses to promote again.

Demotion. For same-application demand, each decay scan considers objects below the coldness threshold H_c for demotion only if their slots have blocked another object's promotion. This preserves temporarily inactive objects when no replacement needs the space. Cross-application demand instead follows the occupancy-based policy in §7.

Delete. Deletion invalidates a fast-tier slot or unlinks a slow-tier node. Removed nodes are not reused until ongoing lookups finish (§6.3).

6 Safe Concurrent Access During Migration

NICMM moves objects between tiers while packet-processing workers continue accessing them. Concurrent movement can expose partial values, cause a lookup to miss an object, or allow access to prematurely reused memory. The challenge is to keep lookups correct during migration without synchronization overhead offsetting fast-tier latency gains.

Per-object locks make lookups wait during updates and migration, adding delay to packet processing. A single lock for all managed state also stalls unrelated lookups.

6.1 Read-Optimistic Concurrency

We propose *read-optimistic concurrency* across tiers to keep lookups fast during migration. Our insight is that fast-tier lookups need only validate reads from permanently allocated slots, while slow-tier lookups must also prevent removed nodes' memory from being reused during a lookup. Inspired by optimistic concurrency control [21], we use per-slot version counters to detect concurrent changes and retry inconsistent fast-tier reads (§6.2). We recheck these counters to detect migration into a slot already checked, preventing an incorrect "not found" result. We defer freeing slow-tier nodes until ongoing lookups finish (§6.3) and serialize slow-tier writers with per-bucket locks (§6.4). This way, fast-tier lookups need no locks, and slow-tier lookups additionally record their activity to delay reclamation.

Note that our target packet-processing workloads are read-heavy, with many writes updating per-flow statistics such as packet and byte counts. Slow-tier readers may observe stale or partially updated values during concurrent writes, which is acceptable for these statistics. Providing stronger consistency for slow-tier value reads is left to future work.

6.2 Lockless Lookups with Version Counters

We adapt version-based lookup techniques (*e.g.,* [14]) to validate returned values and unsuccessful fast-tier checks before reporting an object absent. Each slot's version starts at zero and also supports writer synchronization. Writers acquire exclusive access through a test-and-add loop that retries on contention. The owning writer keeps the version odd while changing the slot's contents and advances it to the next even version when releasing ownership. A lookup reads the counter before accessing the slot and again after copying the fields needed to identify the object and return its value. The lookup accepts the result only if both versions match and are even.

Validating absence across tiers. A lookup can incorrectly report an object absent if the object moves into a tier the lookup has already checked. For example, a lookup finds no matching object in the fast tier, then also finds none in the slow tier because the object was promoted between these checks. Migration makes the destination copy available to lookups before removing the source, but cannot prevent this lookup from missing both copies. The fixed key-to-slot mapping enables *saved-version revalidation* across tiers (Algorithm 1 in §C). If the slow-tier lookup finds no matching object, the worker rechecks the versions saved during its fast-tier checks. Unchanged, even versions mean no object entered those slots while the worker checked the slow tier; otherwise, the lookup restarts. The destination's version also changes on promotion between fast tiers, exposing movement into an already checked slot. Reusing the slots' version counters validates both returned values and "not found" results without adding metadata fields or lookup locks. Before reporting an object absent, revalidation adds one counter read per fast tier.

6.3 Epoch-Based Reclamation

Version validation suffices for fast-tier lookups because their slots remain allocated. Slow-tier lookups also need protection against memory reuse because nodes can be freed. Promoting or deleting a slow-tier object removes its node by updating chain pointers. Later lookups skip it, but an ongoing lookup may still hold its address when the memory is reused. We adapt *epoch-based reclamation* [16]: removed nodes are *retired*, or set aside without reuse, until those lookups finish. Checking every worker on the NIC would make reclamation cost grow with worker count. We exploit flow-based steering: when all lookups for a key stay within one processor group, only that group's workers can hold pointers to its nodes. Its migrator checks these workers before freeing retired nodes.

Tracking active lookups. The group maintains a shared epoch counter; its migrator advances it and checks retired nodes at a configurable epoch-maintenance interval T_e. Before traversing a slow-tier chain, a worker records the current epoch and its active flag in shared memory observed by the migrator. This announcement must be visible to the migrator before the worker reads chain pointers. It clears the flag after completing the lookup and copying any returned value. The runtime removes a node from its chain before retiring it with the group's current epoch, E_r. It is safe to free once every worker is either inactive or active with an epoch snapshot greater than E_r. An inactive worker has finished its traversal; a worker in a later epoch began after the node was removed and could not have reached it.

6.4 Serializing Slow-Tier Writers

We serialize slow-tier insertions, removals, and value updates with one lock per bucket. Insertions and removals change the bucket's first-node pointer or a preceding node's next-node pointer. Concurrent changes to the same pointer can overwrite one another, losing an insertion or undoing a removal. Value updates use the same lock as promotion, preventing updates to a source object while it is copied and removed. Keeping these locks in slow-tier bucket metadata preserves fast-tier slot capacity. Different buckets proceed independently, and lookups acquire no bucket locks. Writers hold the lock while finding and revalidating the node and applying the change, so its cost still depends on chain length.

6.5 Coordinating Promotion and Demotion

We combine version validation, bucket locks, and deferred freeing to keep objects readable during migration.

Inline promotion. For a slow-tier object, NICMM uses the packet-processing worker after egress to acquire its bucket lock, find the object, and increment its access count. If the count reaches the promotion threshold and the candidate fast-tier slot is available (§5.3), NICMM writes the destination under version protection before removing the source from the chain. It releases the bucket lock before submitting the node for epoch-based reclamation, so waiting to enqueue a retired node cannot block a migrator that needs the same bucket. If the candidate is occupied, the object retains its count and stays in the slow tier for a later attempt. Destination selection is constant-time, while source lookup scans the chain.

Background migration and freeing. Each group's migrator advances epochs, frees eligible nodes, and applies placement and sharing policies (§5.3, §7). Fast-tier value updates, deletion, and migration use the same test-and-add acquisition loop to serialize changes to a slot. After acquiring ownership and revalidating the object, the migrator keeps the source version odd throughout copying and invalidation. This prevents competing writers from changing the source during transfer and causes concurrent lookups to retry. The migrator makes the destination copy available to lookups before invalidating the source. If destination allocation fails, it leaves the source intact and releases ownership. Fast-tier lookups only read and validate versions (§6.2). Flow-based steering assigns each fast-tier slot to one migrator.

7 Work-Conserving Fast-Tier Sharing

So far, we have discussed allocation, state access, and safe movement mechanisms needed regardless of the number of applications. With multiple applications, policies must also balance fast-tier utilization and application shares as demand changes. Each sharing participant is an application instance, including tenants running the same application (§2.2.3).

We focus on sharing scarce fast-tier capacity and statically partition the larger slow tier among applications at initialization. Objects are allocated and freed dynamically within each application's partition. If the partition lacks space, new object insertion fails, and a demotion requiring slow-tier storage leaves its source intact.

Hard caps on per-application fast-tier occupancy enforce weighted budgets, but strand capacity when an application does not use its share. Our default fast-tier sharing policy therefore lets applications *borrow* idle capacity beyond their budgets. The challenge is deciding when to reclaim borrowed slots: demoting their objects without competing demand can sacrifice useful fast-tier state.

Demand-triggered sharing. A blocked promotion identifies both the fast-tier slot an object needs and the application holding it. We exploit this signal to propose *demand-triggered sharing*: excess occupancy becomes a reason to yield capacity when it blocks an application below its share. This ties enforcement to unmet demand, allowing useful borrowing to continue while no other application needs the space. Object hotness and excess occupancy then control how aggressively the policy selects demotions, balancing the return of borrowed capacity against retaining useful fast-tier state.

Budgets and ownership. The operator assigns each application a weight w_a. An application owns one or more data structures, whose owner identifiers attribute occupied slots to that application. Its budget in fast tier l is

$$B_{a,l} = \left\lfloor C_l \frac{w_a}{\sum_j w_j} \right\rfloor,$$

where C_l is the tier's slot capacity. The runtime precomputes budgets and maintains per-application occupancy $O_{a,l}$.

Demand signaling. In our set-associative layout, hashing selects both a set and one way, giving each object one candidate slot per tier (§5.2). An empty candidate accepts promotion regardless of the application's budget. When the candidate is occupied, promotion leaves the object in its current tier and marks the slot as in demand. Cross-application demand is recorded only when the requesting application is below budget. Weights define budgets but do not determine priority between the requester and slot owner. The runtime records this demand without moving the occupying object on the packet-processing path.

Occupancy-aware demotion. The migrator processes cross-application demand asynchronously at a configurable interval T_s. For a demanded slot owned by application a, it demotes the object only if $E_{a,l} = O_{a,l} - B_{a,l} > 0$ and $accesses < P_l + KE_{a,l}$. Here, P_l supplies the tier's baseline hotness threshold, including for the fastest tier, which has no promotion destination. The sharing coefficient K balances strict sharing against aggregate hit rate. Higher excess occupancy raises the threshold, making more objects eligible for demotion.

Properties and scope. Because reclamation requires both unmet demand and excess occupancy, applications keep borrowed capacity until others need their shares. Enforcement is asynchronous and depends on object hotness, hash-selected slots, and space for demoted objects in the owner's slow-tier partition. Weighted budgets therefore guide sharing without guaranteeing exact proportional occupancy. This policy governs capacity sharing among trusted applications; it does not isolate processing time or memory bandwidth, or protect against firmware that directly accesses physical memory.

8 Implementation

We implement NICMM in approximately 2.9K lines of Micro-C on the Netronome Agilio NFP-4000, excluding third-party libraries. We will open-source our implementation.

Runtime. The runtime links into packet-processing firmware alongside application logic. CTM and IMEM hold fast-tier arrays sized at build time. EMEM is statically partitioned among applications at initialization, with objects allocated

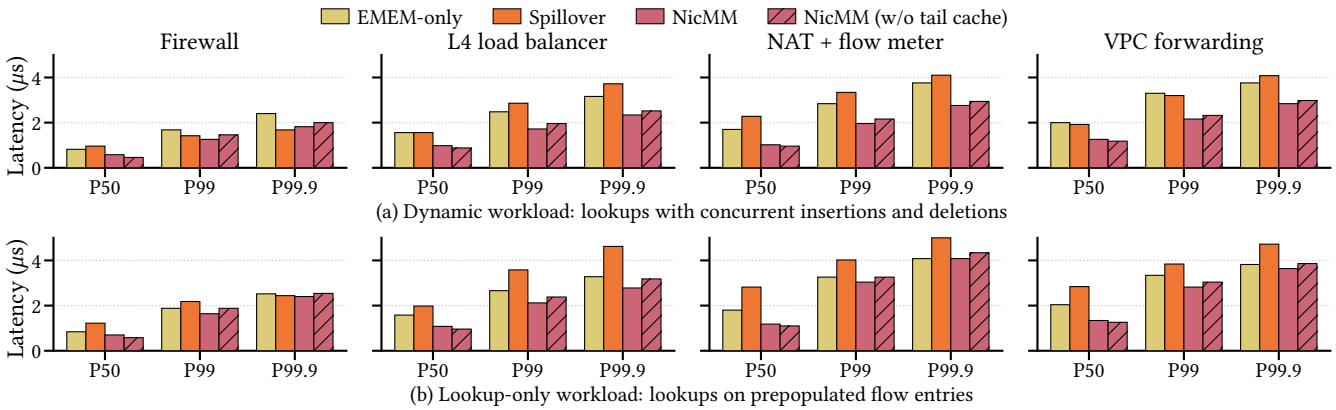


Figure 6: P50, P99, and P99.9 packet processing latency under dynamic and lookup-only workloads with 100K flows. All configurations admit the full working set.

and freed dynamically within each partition. Each island's migrator manages local CTM and an IMEM region aligned with flow steering.

Epoch state, version counters, and access counters reside in island-local memory. Writer acquisition uses atomic test-and-add on the slot's version counter with retry on contention (§6.2). Each worker's active flag and epoch snapshot share a word for reclamation checks. Counters occupy separate words from slot-validity metadata because NFP sub-word writes use word-sized read-modify-writes.

Application API. Applications register key and value sizes at initialization. The API supports lookup, insertion, update, deletion, and counter increment over opaque key-value entries. Table 1 in §B summarizes the calls. Operations accept a key and precomputed hash, reusing packet-steering hashes. After packet egress, the worker records hotness and attempts promotion. §E shows a firewall implemented using the API.

Policy configuration. Promotion thresholds default to $P_{\text{EMEM}} = 4$ and $P_{\text{IMEM}} = 8$ accesses; CTM uses $P_{\text{CTM}} = 4$ as its sharing baseline. Decay uses $T_d = 100$ ms and $\rho = 1/4$ via integer shifts, clearing counts below four. The coldness threshold is $H_c = 1$, and the sharing coefficient is $K = 2$. Sharing and epoch maintenance use nominal intervals $T_s = T_e = 10$ ms with 1 ms migrator polling.

9 Evaluation

Testbed setup. We run applications on a Netronome Agilio CX 40GbE NIC [1] in a server with two 28-core Xeon Gold 6258R processors and 256 GB DRAM. The server runs Ubuntu 20.04.6 LTS and Linux 5.4 for Netronome SDK compatibility. A ConnectX-6 Dx 100GbE NIC on the same host generates traffic over a direct 40 Gbps link to the Agilio NIC.

Applications. Four applications exercise different object sizes, state-access patterns, and packet-header accesses through the NICMM API. The firewall tracks connections; the L4 load balancer maps flows to backends; NAT with flow metering combines header rewriting, translation lookups, and per-packet byte-counter updates. VPC forwarding accesses inner and outer headers and stores each flow's inner destination MAC and outer destination IP. For multi-tenant experiments, we combine the applications into a single firmware image running on every worker. Each worker dispatches packets to the appropriate application by destination IP address.

Workloads. Our DPDK-based traffic generator [13] controls flow arrivals and access concentration. Single-application workloads default to highly skewed, Zipf-distributed flow sizes (packets per flow) with $\alpha = 1.2$. This reflects the skewed accesses observed in our data-center traces (§2.2.2). Sensitivity experiments use $\alpha \in \{1.2, 1.0, 0.8\}$ and uniform accesses. Latency experiments use 64-byte packets (128 bytes for VPC); throughput experiments vary packet size. Each experiment specifies flow populations, arrival rates, and deviations from these defaults.

Baselines. We implement CTM-, IMEM-, and EMEM-only baselines in Micro-C. Among them, we focus on EMEM-only because it holds the entire working set. Our *Spillover* baseline tries CTM, then IMEM, then EMEM, inserting into the first tier with available space. It never relocates admitted objects, so placement follows arrival order and available space without adapting to hotness. To isolate the effect of the tail cache (§5.2), we also evaluate NICMM without it.

9.1 Per-Packet Processing Latency

Each application processes 100K flows, exceeding fast-tier capacity, under two workloads. The *dynamic* workload interleaves lookups with flow insertions and deletions. The *prepopulated* workload issues only lookups after populating the table. Both use the default access distribution and packet sizes. We measure processing latency from packet ingress to immediately before egress.

NICMM dynamically places frequently accessed entries in CTM or IMEM, allowing common lookups to avoid EMEM even when the full working set exceeds fast-tier capacity. This lowers median latency across all four applications in both workloads (Figure 6): by 29–40% relative to EMEM-only placement with dynamic flows and 17–34% with prepopulated

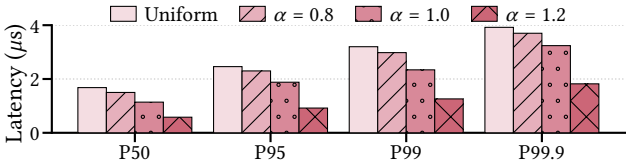


Figure 7: NICMM firewall packet processing latency *vs.* skew.

state. Unlike CTM- and IMEM-only placement, NICMM admits the full working set.

Spillover uses multiple tiers, but its benefit depends on whether insertion order places hot objects in fast tiers. It can leave hot objects in EMEM while cold objects occupy fast-tier slots. By adapting placement to accesses, NICMM reduces median latency relative to Spillover by 34–55% with dynamic flows and 43–58% with prepopulated state, while both admit the full workload.

Tail latency also generally improves, though lookups for entries remaining in EMEM still incur slow-tier accesses. P99 is lower than EMEM-only placement in every case, and P99.9 is lower than both baselines in most cases. Firewall P99.9 is modestly higher than Spillover under dynamic flows. Disabling the tail cache reduces median latency by 6–21% but increases P99 and P99.9 latency by 5–16% across all applications and both workloads. We enable the tail cache throughout the remaining experiments.

Sensitivity to access skew. We measure firewall latency with 100K flows under $\alpha = 1.2$, $\alpha = 1.0$, $\alpha = 0.8$, and uniform accesses, holding fast-tier capacity fixed. P99.9 rises from 1.82 to 3.24 to 3.7 and 3.92 μs, respectively; other percentiles follow the same trend (Figure 7). As accesses spread across more objects, a fixed fast tier covers a smaller fraction of accesses. Other applications show similar trends.

9.2 Packet Processing Throughput

We prepopulate each application with 100K flows and send traffic at 40 Gbps with the skewed workload ($\alpha = 1.2$). Packet sizes range from 64 to 1,518 bytes (128 to 1,518 bytes for VPC). We report receiver throughput against EMEM and the Spillover baseline, which admit all flows. Each experiment is repeated ten times; we report the mean of the per-run median throughput, with error bars showing the sample standard deviation across runs.

NICMM delivers these latency benefits with comparable throughput, staying within 7% of EMEM across all applications and packet sizes and exceeding EMEM in three of the six configurations below line rate (Figure 8). Both systems reach 40 Gbps at every tested size of at least 256 bytes. Smaller packets require higher packet rates, increasing processing demand; both systems then fall below line rate at 64 bytes and, for all applications but the firewall, at 128 bytes. NICMM's largest shortfall is 6.7% for 64-byte NAT packets (19.91 Gbps *vs.* 21.34 Gbps). We attribute this gap to longer slow-tier chains, as the tail cache's fast-tier space budget limits the number of buckets. NICMM matches or exceeds Spillover in all other configurations.

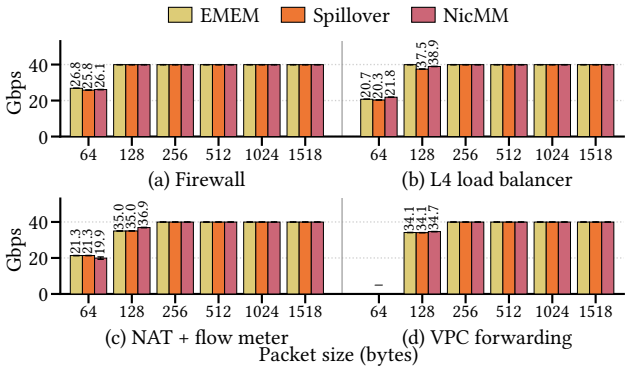


Figure 8: Receiver throughput under 40 Gbps offered load. VPC uses packets of at least 128 bytes.

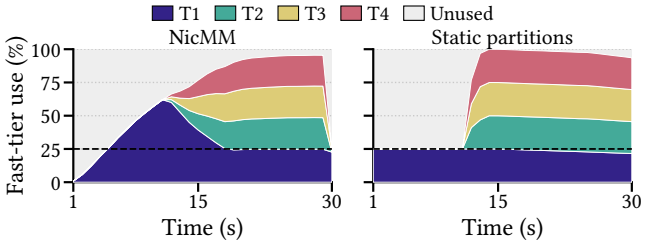


Figure 9: Fast-tier occupancy by tenant (gray: unused), normalized to each system's capacity. Dashed lines mark T1's 25% share. Traces start at first nonzero occupancy.

9.3 Multi-Tenant Memory Sharing

We revisit both scenarios in §2.2.3: partitioned tables across applications and a shared table within one application.

Sharing across applications. Four equally weighted tenants run highly skewed workloads: a firewall (T1), VPC forwarding (T2), NAT (T3), and a load balancer (T4). Each has a 25% target share; T1 starts about 10 seconds before the others. We compare NICMM's 24,576 shared CTM/IMEM slots against 16,384 CTM slots statically partitioned among tenants. Figure 9 normalizes occupancy by each system's capacity.

T1 borrows idle capacity, peaking near 60% as the others become active; static partitions hold it at 25%. As other tenants demand capacity, NICMM demotes T1's objects, returning its occupancy to approximately 25% within seven seconds of their arrival.

Sharing within one application. Two equally weighted firewall tenants share a table; T2 starts ten seconds earlier and has twice T1's arrival rate. NICMM has 24,576 fast-tier slots; the shared CTM baseline has 32,768. Shared CTM inserts flows directly into the fast tier; NICMM promotes them from the slow tier after sufficient accesses. T2's lower initial occupancy reflects promotion warm-up, not share enforcement.

After T1 starts at 10 s, its blocked promotions reclaim T2's borrowed slots, bringing T2 toward its 50% share (Figure 10). Hash-derived candidates allow contention despite free slots elsewhere. Without enforcement, shared CTM leaves T2 occupying most capacity.

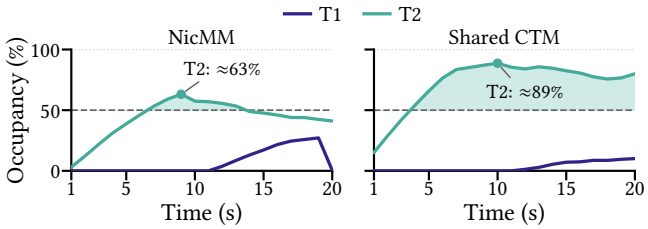


Figure 10: Fast-tier occupancy for two firewall tenants (T2's arrival rate: 2× T1's), normalized per system. Shading marks T2's excess over its 50% target (dashed).

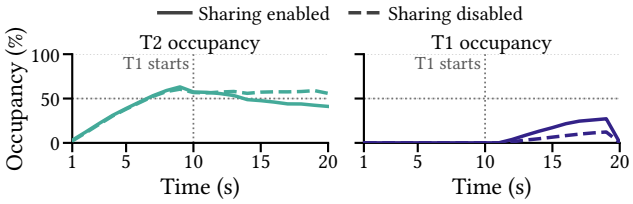


Figure 11: Fast-tier occupancy with sharing enabled and disabled in NICMM, normalized to its capacity. Dotted lines mark each tenant's 50% target and T1's arrival at 10 s.

Effect of demand-triggered sharing. Figure 11 reuses the NICMM traces from Figure 10 and repeats the experiment with sharing disabled, keeping capacity, promotion thresholds, and workload fixed. Both configurations initially promote T2's objects and let it borrow beyond 50%. T1's occupancy also rises only after its objects accumulate enough accesses for promotion. Eight seconds after T1 starts, it occupies 26% of the fast tier with sharing enabled *vs.* 11% without it, while T2 occupies 44% *vs.* 58%, respectively. Reclaiming T2's borrowed slots thus lets more of T1's hot state reach fast tiers, while preserving borrowing before competing demand arises.

9.4 Microbenchmarks

To complement our end-to-end evaluation, we conduct microbenchmarks to evaluate the effectiveness and overheads of NICMM's key components.

9.4.1 Effectiveness of Hierarchical Allocation

We compare backend-only, middle-end plus backend, and full allocation to isolate each layer's benefits (§4).

Scaling under contention. We scale a NIC port of the GLIBC benchmark [4] to 480 workers. Each worker allocates and frees sixteen 8-byte objects from EMEM three times, contending on one size class's middle-end sectors and shared frontend ring. We average the slowest worker's completion time across three runs and normalize to backend-only allocation at the same worker count.

At 480 workers, adding the middle-end yields a 39.8× speedup; the full allocator reaches 435× (Figure 12). The middle-end serves many allocations per backend request, reducing contention on the shared backend lock; the frontend reuses freed blocks without acquiring the middle-end lock.

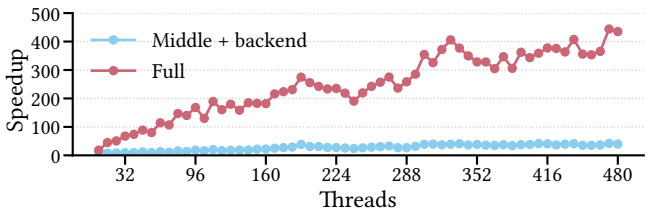


Figure 12: Speedup over backend-only at equal worker counts.

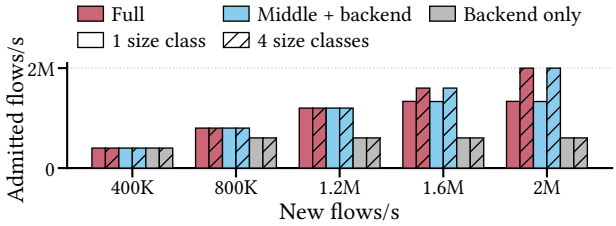


Figure 13: Median flow admission rate with one or four size classes. Full includes frontend, middle-end, and backend.

Packet-driven allocation throughput. We measure the allocation throughput by calling malloc on receiving a SYN packet. With 192 active workers, we offer 400K–2M new flows/s for 5 seconds without deletions, using one or four size classes, and report median admitted flows/s. Backend-only allocation saturates at 604K flows/s with either size-class count because all allocations share one lock (Figure 13). The middle-end amortizes backend requests, admitting 1.33M flows/s with one size class and 2M with four. Each size class has its own middle-end lock, so using more classes spreads allocations across locks and reduces contention. The frontend adds no benefit in this insertion-only workload: without flow deletions, there are no freed blocks to reuse.

9.4.2 Effectiveness of Tiered Lookup

Independence from slow-tier chains. We compare NICMM with a chained hash table (CHT) that promotes objects without reordering chains. Each bucket holds two or four firewall flows; we promote and look up only the last flow.

Doubling chain length raises CHT's P99 from 1.44 to 1.90 μs; NICMM stays at 0.86 μs (Figure 14). Direct key-based lookup avoids CHT's preceding slow-tier accesses (§5).

Steady-state lookup throughput. We compare throughput when lookups hit CTM, IMEM, or EMEM with NICMM's overall throughput under dynamic placement. Each configuration holds the same 4,096 prepopulated firewall flows; single-tier configurations disable demotion so lookups hit the selected tier. We send 64- and 128-byte packets at 40 Gbps for 5 seconds and report median receiver throughput. With 64-byte packets, NICMM achieves 26.76 Gbps, comparable to CTM-only (26.73) and IMEM-only (26.63), and 15% above EMEM-only (23.28). All reach 40 Gbps at 128 bytes.

9.4.3 Effectiveness of Lockless Lookup

We compare NICMM's version-based lookup (§6.2) against locking each accessed slot, using firewall workloads with $\alpha = 1.2$, $\alpha = 0.8$, and uniform accesses.

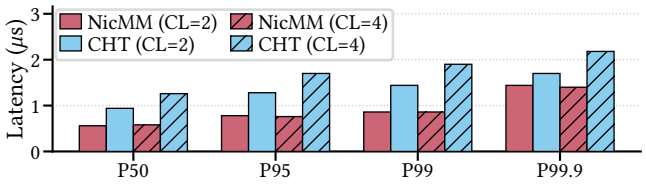


Figure 14: Hot-flow lookup latency *vs.* a chained hash table (CHT); CL denotes chain length.

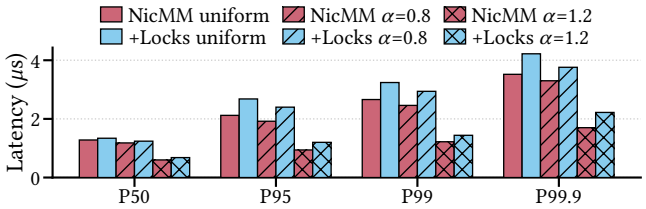


Figure 15: Firewall lookup latency with version validation *vs.* per-slot read locks.

Locking raises P50 by 5–13% and P99.9 by 14–31% (Figure 15); at $\alpha = 1.2$, P99.9 rises from 1.70 to 2.22 μs. Skew concentrates reads on hot slots, creating more opportunities for lock contention; more fast-tier hits also make synchronization a larger fraction of lookup cost. The largest relative P99.9 penalty at $\alpha = 1.2$ is consistent with these effects. Reads run concurrently but may retry when writes overlap.

9.4.4 Resource Overheads

Memory. Fast-tier metadata uses 24 bytes per slot across 8,192 CTM and 16,384 IMEM slots, totaling 576 KiB—about 11% of the 5 MiB combined CTM and IMEM capacity used by NICMM. For 2,048-byte slow-tier sectors, metadata uses 16 bytes (0.8%) per backend sector and 48 bytes (2.3%) per middle-end sector. Sector-level tracking amortizes metadata across objects without per-worker free-object caches.

Instruction memory. The runtime occupies 5,585 of each FPC's 8,192 instruction slots, including 1,894 for the allocator, leaving 2,607 slots (31.8%) for application code. The remaining space accommodates the packet-processing logic of the applications alongside the full runtime.

10 Discussion

Supporting a caching model. We evaluate tiering, which uses the tiers' combined capacity for distinct objects. NICMM could also support a *write-back cache* with a slow-tier backing entry per object and at most one fast-tier copy outside transfers. Eviction would write back dirty values or discard clean copies, avoiding slow-tier allocation during eviction at the cost of duplicate storage. Transfers between fast tiers would preserve dirty state without a slow-tier write-back.

Caching could reuse NICMM's object operations, hotness and sharing signals, and policy interface, but requires dirty-state tracking and coordinated updates, fills, and write-back. Version validation and deferred freeing would remain useful. Before returning a backing value, lookups would also revalidate fast-tier misses to detect a modified cached copy moving into an already-checked tier.

Portability to other NIC architectures. On FPGA-based NICs [2, 9], NICMM could be realized as a reusable *memory-management hardware block* shared by packet-processing pipelines. These NICs expose on-chip block RAM/UltraRAM and larger DRAM or HBM [2], but these resources alone do not manage object allocation, placement, or sharing. Hardware engines could implement key-based object operations, while a background engine migrates objects according to access counts, demand signals, and configurable policies. Programmable ASIC NICs such as AMD Pensando [3] and Intel's E2100 IPU [5] expose architecture-specific packet-processing interfaces. Porting NICMM depends on their support for the required memory operations and background migration, and may require vendor extensions or hardware changes.

11 Related Work

Multi-tenancy support for NICs. FairNIC [17] combines core partitioning, cache striping, and accelerator rate limiting to isolate tenants. PANIC [23] and SuperNIC [24] support shared offload chains, while OSMOSIS [18] dynamically multiplexes compute and I/O with tenant-level QoS. S-NIC [30] provides hardware-enforced security isolation across NIC resources. Our focus is *dynamic* memory allocation, placement, and sharing across colocated NIC applications.

FPGA multi-tenancy. AmorphOS [19] multiplexes reconfigurable tasks in space and time, and ViTAL [28] decouples application compilation from runtime fabric allocation. Coyote [20] provides virtual memory, communication, and memory management in cooperation with the host OS. NICMM targets object allocation and placement across NIC memory tiers, where management overhead must fit the latency and resource budgets of packet processing.

Host memory tiering. TPP [25] adapts page placement across local and CXL memory, while Memtis [22] adapts placement and page size. OBASE [10] groups objects by hotness for OS page tiering. AIFM [27] supports application-integrated object movement between local and remote memory. Database and storage indexes also use record-level tiering [26, 29]. Our setting is shared, software-managed NIC memory without OS virtual-memory support.

12 Conclusions

We presented NICMM, a memory management runtime for on-path SmartNICs. On-path NICs are essentially bare-metal computers without an OS, leaving developers to manage shared hardware resources alongside packet-processing logic. Memory is one instance of this broader systems challenge. By managing allocation, placement, and sharing through a key-value API, NICMM lets memory use adapt to changing workloads transparently to applications. This separation makes memory management a reusable service whose policies can evolve independently of application logic. We envision future

NIC runtimes extending this approach to other resources, giving developers a common foundation for building applications without repeatedly implementing the mechanisms needed to manage and share on-NIC resources.

References

[1] Agilio CX SmartNICs - Netronome. https://www.netronome.com/products/agilio-cx/.

[2] Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[3] AMD Pensando Infrastructure Accelerators. https://www.amd.com/en/accelerators/pensando.

[4] GLIBC bench-malloc-simple.c. https://github.com/daanx/mimalloc-bench/blob/master/bench/glibc-bench/bench-malloc-simple.c.

[5] Intel IPU E2100. https://www.intel.com/content/www/us/en/content-details/818147/intel-infrastructure-processing-unit-intel-ipu-soc-e2100-product-brief.html.

[6] Intel® Infrastructure Processing Unit (Intel® IPU) Platform F2000X-PL. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/f2000x-pl-platform.html.

[7] Pensando DSC-25 Distributed Services Card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf.

[8] TCMalloc : Thread-Caching Malloc. https://github.com/google/tcmalloc/blob/master/docs/design.md. Accessed 2025-07-15.

[9] The Alveo U55C Data Center Accelerator Card. https://www.xilinx.com/applications/data-center/high-performance-computing/u55c.html.

[10] Vinay Banakar, Suli Yang, Kan Wu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Kimberly Keeton. OBASE: Object-Based Address-Space engineering to improve memory tiering. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pages 151–166, 2026.

[11] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1469–1487, 2023.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference (IMC 10)*, pages 267–280, 2010.

[13] DPDK Project. DPDK: Data plane development kit. https://www.dpdk.org/about/.

[14] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[15] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[16] Keir Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[17] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC performance isolation with FairNIC. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 20)*, pages 681–693, 2020.

[18] Mikhail Khalilov, Marcin Chrapek, Siyuan Shen, Alessandro Vezzu, Thomas Benz, Salvatore Di Girolamo, Timo Schneider, Daniele De Sensi, Luca Benini, and Torsten Hoefler. OSMOSIS: Enabling multi-tenancy in datacenter SmartNICs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 247–263, 2024.

[19] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, 2018.

[20] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010, 2020.

[21] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[22] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP 23)*, pages 17–34, 2023.

[23] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 243–259, 2020.

[24] Will Lin, Yizhou Shan, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. SuperNIC: An FPGA-Based, cloud-oriented SmartNIC. In *Proceedings of the 2024 ACM/SIGDA*

International Symposium on Field Programmable Gate Arrays (FPGA 24), pages 130–141, 2024.

[25] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 23)*, pages 742–755, 2023.

[26] Jiansheng Qiu, Fangzhou Yuan, Mingyu Gao, and Huanchen Zhang. HotRAP: Hot record retention and promotion for LSM-trees with tiered storage. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pages 497–511, 2025.

[27] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[28] Yue Zha and Jing Li. Virtualizing FPGAs in the cloud. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 20)*, pages 845–858, 2020.

[29] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Two is better than one: The case for 2-Tree for skewed data sets. In *13th Annual Conference on Innovative Data Systems Research (CIDR 23)*, 2023.

[30] Yang Zhou, Mark Wilkening, James Mickens, and Minlan Yu. SmartNIC security isolation in the cloud with S-NIC. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 24)*, pages 851–869, 2024.

A NFP-4000 Architecture

Figure 16 shows five general-purpose islands, each with 12 FPCs, for 60 FPCs in total. Each island also contains local CTM and cluster-local storage (CLS). The islands share IMEM and off-chip EMEM. NICMM uses CTM and IMEM as fast tiers and EMEM as the slow tier, while CLS holds synchronization metadata such as version counters and worker epoch state.

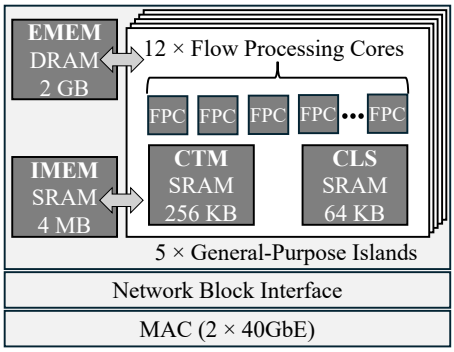


Figure 16: NFP-4000 architecture. Dark grey blocks indicate compute and memory resources.

B Application API Reference

Table 1 summarizes the core object operations. See §E for a firewall example and runtime setup.

API Call	Purpose
`kvstore_lookup(key)`	Read an object's value.
`kvstore_insert(key, value)`	Create an object.
`kvstore_update(key, value)`	Replace an object's value.
`kvstore_delete(key)`	Delete an object.
`kvstore_increment (key, offset, delta)`	Add `delta` to a 32-bit counter at byte `offset` within the value.
`kvstore_record_access(key)`	Record hotness and attempt promotion after packet egress.

Table 1: Core NICMM APIs with simplified parameters. Store handles, hashes, output buffers, types, and memory qualifiers are omitted.

C Lookup with Saved-Version Revalidation

Algorithm 1 details the base lookup procedure described in §6.2. It validates each fast-tier probe and, before reporting an object absent, rechecks saved versions to detect migration into a tier already searched.

Algorithm 1: Lookup with saved-version revalidation.

Input: Owner identifier d and key k

retry:
foreach *fast tier l, from fastest to slowest* **do**
  $s \leftarrow \text{candidate}(k, l)$
  $v_1 \leftarrow \text{version}(s)$
  $x \leftarrow \text{copy}(s)$ *// identity fields and value*
  $v_2 \leftarrow \text{version}(s)$
  if *v_1 is odd or $v_1 \neq v_2$* **then go to retry**
  if *x is valid and matches (d, k)* **then return** x.value
  $\text{saved}[l] \leftarrow v_2$ *// retain the validated miss*
$r \leftarrow \text{slowLookup}(d, k)$ *// epoch-protected search*
if *r is found* **then return** r.value
foreach *fast tier l* **do**
  $v \leftarrow \text{version}(\text{candidate}(k, l))$
  if $v \neq \text{saved}[l]$ **then go to retry**
return NOTFOUND

D Fast-Tier Slot Layout

Figure 17 illustrates the logical fields associated with the fast-tier slots described in §5.2.

Version (4B)	Access count (4B)	Valid (1B)	Owner (1B)	Hash (4B)	Payload (Key-value pair)

Figure 17: Logical fields associated with a fast-tier slot.

E Firewall Object Lifecycle in Micro-C

Figure 18 shows a simplified LAN-side worker from our connection-tracking firewall. All contexts initialize NICMM and obtain a key-value store handle; one context per island runs the migrator. A SYN creates a state object, subsequent packets look it up, and a FIN deletes it. After packet egress, the worker records successful lookups so NICMM can track hotness and promote objects. The application identifies state by a five-tuple and a precomputed hash; NICMM handles allocation, placement, migration, and reclamation. WAN-side processing, omitted here, looks up the same key with the source and destination endpoints reversed.

The listing retains the implementation's API names and Micro-C memory qualifiers. The helpers `receive_lan_packet` and `send_packet` abbreviate packet I/O, key construction (including zeroed padding), and result handling; they are not NICMM APIs. The initial value buffer is in `__emem` because insertion takes a `__mem40` source pointer and copies its contents into managed storage.

```c
#include <nfp.h>
#include <nfp/me.h>
#include <nfp6000/nfp_me.h>
#include <stdint.h>
#include <net/tcp.h>
#include "kvstore.h"
typedef __packed struct {
    uint32_t src_ip, dst_ip;
    uint16_t sport, dport;
    uint8_t proto, padding[3];
} fw_key_t;
__emem uint8_t initial_state[4] = {0, 0, 0, 1};
int main(void) {
    uint16_t weight = 1;
    uint16_t key_size = 16, value_size = 4;
    __lmem kvstore_t *fw;
    __lmem fw_key_t key;
    __lmem uint32_t state;
    uint32_t hash, result, hit;
    uint8_t flags;
    // All contexts initialize the runtime.
    nicmm_init(&weight, &key_size, &value_size);
    fw = kvstore_get_handle(0);
    // One context per island runs maintenance
    // forever; other contexts process packets.
    if (__MENUM == 0 &&
        ctx() == (__nctx_mode() == 4 ? 6 : 7))
        kvstore_start_migrator();
    for (;;) {
        receive_lan_packet(&key, &hash, &flags);
        hit = 0;
        if (flags & NET_TCP_FLAG_SYN) {
            // Create connection state.
            result = kvstore_insert(
                fw, hash, &key,
                (__mem40 void *)initial_state);
        } else if (flags & NET_TCP_FLAG_FIN) {
            // Delete state; NicMM reclaims storage.
            result = kvstore_delete(fw, hash,
                                    &key, 0);
        } else {
            // Read state without knowing its tier.
            hit = kvstore_lookup(fw, hash,
                                 &key, &state);
            result = hit;
        }
        send_packet(result); // Packet egress.
        if (hit)
            kvstore_record_access(fw, hash,
                                  &key);
    }
}
```

Figure 18: Simplified Micro-C firewall using NICMM APIs (highlighted) to create, access, and delete connection state. Hotness recording follows packet egress; memory movement remains inside the runtime.