MICA: A Holistic Approach to Fast In-Memory Key-Value Storage

Hyeontaek Lim,[1] Dongsu Han,[2] David G. Andersen,[1] Michael Kaminsky[3]
[1]*Carnegie Mellon University,* [2]*KAIST,* [3]*Intel Labs*

Abstract

MICA is a scalable in-memory key-value store that handles 65.6 to 76.9 million key-value operations per second using a single general-purpose multi-core system. MICA is over 4–13.5x faster than current state-of-the-art systems, while providing consistently high throughput over a variety of mixed read and write workloads.

MICA takes a holistic approach that encompasses all aspects of request handling, including parallel data access, network request handling, and data structure design, but makes unconventional choices in each of the three domains. First, MICA optimizes for multi-core architectures by enabling parallel access to partitioned data. Second, for efficient parallel data access, MICA maps client requests directly to specific CPU cores at the server NIC level by using client-supplied information and adopts a light-weight networking stack that bypasses the kernel. Finally, MICA's new data structures—circular logs, lossy concurrent hash indexes, and bulk chaining—handle both read- and write-intensive workloads at low overhead.

1 Introduction

In-memory key-value storage is a crucial building block for many systems, including popular social networking sites (e.g., Facebook) [36]. These storage systems must provide high performance when serving many small objects, whose total volume can grow to TBs and more [5].

While much prior work focuses on high performance for read-mostly workloads [15, 30, 32, 37], in-memory key-value storage today must also handle write-intensive workloads, e.g., to store frequently-changing objects [2, 5, 36]. Systems optimized only for reads often waste resources when faced with significant write traffic; their inefficiencies include lock contention [32], expensive updates to data structures [15, 30], and complex memory management [15, 32, 36].

In-memory key-value storage also requires low-overhead network communication between clients and servers. Key-value workloads often include a large number of small key-value items [5] that require key-value storage to handle short messages efficiently. Systems using standard socket I/O, optimized for bulk communication, incur high network stack overhead at both kernel- and user-level. Current systems attempt to batch requests at the client to amortize this overhead, but batching increases latency, and large batches are unrealistic in large cluster key-value stores because it is more difficult to accumulate multiple requests being sent to the same server from a single client [36].

MICA (Memory-store with Intelligent Concurrent Access) is an in-memory key-value store that achieves high throughput across a wide range of workloads. MICA can provide either store semantics (no existing items can be removed without an explicit client request) or cache semantics (existing items may be removed to reclaim space for new items). Under write-intensive workloads with a skewed key popularity, a single MICA node serves 70.4 million small key-value items per second (Mops), which is 10.8x faster than the next fastest system. For skewed, read-intensive workloads, MICA's 65.6 Mops is at least 4x faster than other systems even after modifying them to use our kernel bypass. MICA achieves 75.5–76.9 Mops under workloads with a uniform key popularity. MICA achieves this through the following techniques:

Fast and scalable parallel data access: MICA's data access is fast and scalable, using data partitioning and exploiting CPU parallelism within and between cores. Its EREW mode (Exclusive Read Exclusive Write) minimizes costly inter-core communication, and its CREW mode (Concurrent Read Exclusive Write) allows multiple cores to serve popular data. MICA's techniques achieve consistently high throughput even under skewed workloads, one weakness of prior partitioned stores.

Network stack for efficient request processing: MICA interfaces with NICs directly, bypassing the kernel, and uses client software and server hardware to direct remote key-value requests to appropriate cores where the requests can be processed most efficiently. The network stack achieves zero-copy packet I/O and request processing.

New data structures for key-value storage: New memory allocation and indexing in MICA, optimized for store and cache separately, exploit properties of key-value workloads to accelerate write performance with simplified memory management.

2 System Goals

In this section, we first clarify the non-goals and then discuss the goals of MICA.

Non-Goals: We do not change the *cluster* architecture. It can still shard data and balance load across nodes, and perform replication and failure recovery.

We do not aim to handle large items that span multiple packets. Most key-value items will fit comfortably in a single packet [5]. Clients can store a large item in a traditional key-value system and put a pointer to that system in MICA. This only marginally increases total latency; one extra round-trip time for indirection is smaller than the transfer time of a large item sending multiple packets.

We do not strive for durability: All data is stored in DRAM. If needed, log-based mechanisms such as those from RAMCloud [37] would be needed to allow data to persist across power failures or reboots.

MICA instead strives to achieve the following goals:

High single-node throughput: Sites such as Facebook replicate some key-value nodes purely to handle load [36]. Faster nodes may reduce cost by requiring fewer of them overall, reducing the cost and overhead of replication and invalidation. High-speed nodes are also more able to handle load spikes and popularity hot spots. Importantly, using fewer nodes can also reduce job latency by reducing the number of servers touched by client requests. A single user request can create more than 500 key-value requests [36], and when these requests go to many nodes, the time until all replies arrive increases, delaying completion of the user request [10]. Having fewer nodes reduces fan-out, and thus, can improve job completion time.

Low end-to-end latency: The end-to-end latency of a remote key-value request greatly affects performance when a client must send back-to-back requests (e.g., when subsequent requests are dependent). The system should minimize both local key-value processing latency and the number of round-trips between the client and server.

Consistent performance across workloads: Real workloads often have a Zipf-distributed key popularity [5], and it is crucial to provide fast key-value operations regardless of skew. Recent uses of in-memory key-value storage also demand fast processing for write-intensive workloads [2, 36].

Handle small, variable-length key-value items: Most key-value items are small [5]. Thus, it is important to process requests for them efficiently. Ideally, key-value request processing over the network should be as fast as packet processing in software routers—40 to 80 Gbps [12, 19]. Variable-length items require careful memory management to reduce fragmentation that can waste substantial space [5].

Key-value storage interface and semantics: The system must support standard single-key requests (e.g., `GET(key)`, `PUT(key,value)`, `DELETE(key)`) that are common in systems such as Memcached. In cache mode, the system performs automatic cache management that may evict stored items at its discretion (e.g., LRU); in store mode, the system must not remove any stored items without clients' permission while striving to achieve good memory utilization.

Commodity hardware: Using general-purpose hardware reduces the cost of development, equipment, and operation. Today's server hardware can provide high-speed I/O [12, 22], comparable to that of specialized hardware such as FPGAs and RDMA-enabled NICs.

Although recent studies tried to achieve some of these goals, none of their solutions comprehensively address them. Some systems achieve high throughput by supporting only small fixed-length keys [33]. Many rely on client-based request batching [15, 30, 33, 36] to amortize high network I/O overhead, which is less effective in a large installation of key-value stores [14]; use specialized hardware, often with multiple client-server round-trips and/or no support for item eviction (e.g., FPGAs [7, 29], RDMA-enabled NICs [35]); or do not specifically address remote request processing [45]. Many focus on uniform and/or read-intensive workloads; several systems lack evaluation for skewed workloads [7, 33, 35], and some systems have lower throughput for write-intensive workloads than read-intensive workloads [30]. Several systems attempt to handle memory fragmentation explicitly [36], but there are scenarios where the system never reclaims fragmented free memory, as we describe in the next section. The fast packet processing achieved by software routers and low-overhead network stacks [12, 19, 20, 41, 43] set a bar for how fast a key-value system *might* operate on general-purpose hardware, but do not teach how their techniques apply to the higher-level processing of key-value requests.

3 Key Design Choices

Achieving our goals requires rethinking how we design *parallel data access*, the *network stack*, and *key-value data structures*. We make an unconventional choice for each; we discuss how we overcome its potential drawbacks to achieve our goals. Figure 1 depicts how these components fit together.

3.1 Parallel Data Access

Exploiting the parallelism of modern multi-core systems is crucial for high performance. The most common access models are concurrent access and exclusive access:

Concurrent access is used by most key-value systems [15, 30, 36]. As in Figure 2 (a), multiple CPU cores can access the shared data. The integrity of the data structure must be maintained using mutexes [36], optimistic locking [15, 30], or lock-free data structures [34].

Unfortunately, concurrent writes scale poorly: they incur frequent cache line transfer between cores, because only one core can hold the cache line of the same memory location for writing at the same time.

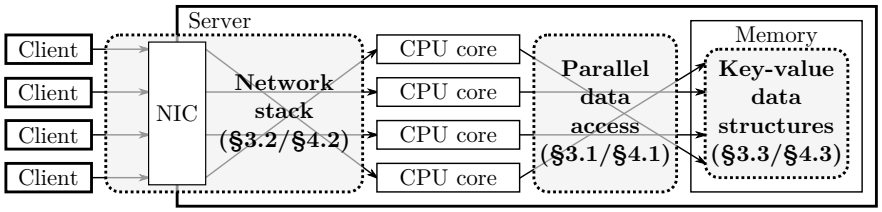
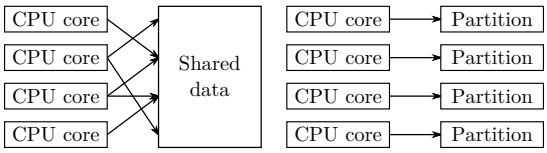


Figure 1: Components of in-memory key-value stores. MICA's key design choices in §3 and their details in §4.



(a) Concurrent access (b) Exclusive access

Figure 2: Parallel data access models.

Exclusive access has been explored less often for key-value storage [6, 25, 33]. Only one core can access part of the data, as in Figure 2 (b). By partitioning the data ("sharding"), each core exclusively accesses its own partition in parallel without inter-core communication.

Prior work observed that partitioning *can* have the best throughput and scalability [30, 45], but cautions that it lowers performance when the load between partitions is imbalanced, as happens under skewed key popularity [15, 30, 45]. Furthermore, because each core can access only data within its own partition, *request direction* is needed to forward requests to the appropriate CPU core.

MICA's parallel data access: MICA partitions data and mainly uses exclusive access to the partitions. MICA exploits CPU caches and packet burst I/O to disproportionately speed more loaded partitions, nearly eliminating the penalty from skewed workloads. MICA can fall back to concurrent reads if the load is extremely skewed, but avoids concurrent writes, which are always slower than exclusive writes. Section 4.1 describes our data access models and partitioning scheme.

3.2 Network Stack

This section discusses how MICA avoids network stack overhead and directs packets to individual cores.

3.2.1 Network I/O

Network I/O is one of the most expensive processing steps for in-memory key-value storage. TCP processing alone may consume 70% of CPU time on a many-core optimized key-value store [33].

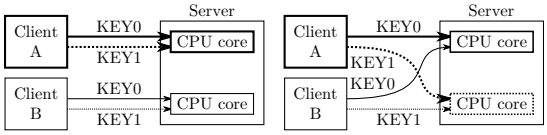
The **socket I/O** used by most in-memory key-value stores [15, 30, 33, 45] provides portability and ease of development. However, it underperforms in packets per second because it has high per-`read()` overhead. Many systems therefore often have clients include a batch of requests in a single larger packet to amortize I/O overhead.

Direct NIC access is common in software routers to achieve line-rate packet processing [12, 19]. This raw access to NIC hardware bypasses the kernel to minimize the packet I/O overhead. It delivers packets in bursts to efficiently use CPU cycles and the PCIe bus connecting NICs and CPUs. Direct access, however, precludes useful TCP features such as retransmission, flow control, and congestion control.

MICA's network I/O uses direct NIC access. By targeting only small key-value items, it needs fewer transport-layer features. Clients are responsible for retransmitting packets if needed. Section 4.2 describes such issues and our design in more detail.

3.2.2 Request Direction

Request direction delivers client requests to CPU cores for processing.[1] Modern NICs can deliver packets to specific cores for load balancing or core affinity using hardware-based packet classification and multi-queue support.



(a) Flow-level core affinity (b) Object-level core affinity

Figure 3: Request direction mechanisms.

Flow-level core affinity is available using two methods: Receive-Side Scaling (RSS) [12, 19] sends packets to cores based by hashing the packet header 5-tuple to identify which RX queue to target. Flow Director (FDir) [41] can more flexibly use different parts of the packet header plus a user-supplied table to map header values to RX queues. Efficient network stacks use affinity to reduce inter-core contention for TCP control blocks [20, 41].

Flow affinity reduces only *transport layer* contention, not *application-level* contention [20], because a single transport flow can contain requests for any objects (Figure 3 (a)). Even for datagrams, the benefit of flow affinity is small due to a lack of locality across datagrams [36].

Object-level core affinity distributes requests to cores based upon the application's partitioning. For example, requests sharing the same key would all go to the core handling that key's partition (Figure 3 (b)).

[1] Because we target small key-value requests, we will use requests and packets interchangeably.

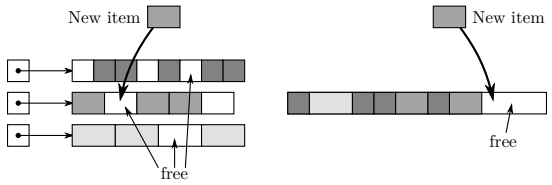
Systems using exclusive access require object-level core affinity, but commodity NIC hardware cannot directly parse and understand application-level semantics. Software request redirection (e.g., message passing [33]) incurs inter-core communication, which the exclusive access model is designed to avoid.

MICA's request direction uses Flow Director [23, 31]. Its *clients* then encode object-level affinity information in a way Flow Director can understand. Servers, in turn, inform clients about the object-to-partition mapping. Section 4.2 describes how this mechanism works.

3.3 Key-Value Data Structures

This section describes MICA's choice for two main data structures: allocators that manage memory space for storing key-value items and indexes to find items quickly.

3.3.1 Memory Allocator



(a) Dynamic object allocator (b) Append-only log structure

Figure 4: Memory allocators.

A **dynamic object allocator** is a common choice for storing variable-length key-value items (Figure 4 (a)). Systems such as Memcached typically use a slab approach: they divide object sizes into classes (e.g., 48-byte, 56-byte, ..., 1-MiB[2]) and maintain separate ("segregated") memory pools for these classes [15, 36]. Because the amount of space that each class uses typically varies over time, the systems use a global memory manager that allocates large memory blocks (e.g., 1 MiB) to the pools and dynamically rebalances allocations between classes.

The major challenge for dynamic allocation is the memory fragmentation caused when blocks are not fully filled. There may be no free blocks or free objects for some size classes while blocks from other classes are partly empty after deletions. Defragmentation packs objects of each object tightly to make free blocks, which involves expensive memory copy. This process is even more complex if the memory manager performs rebalancing concurrently with threads accessing the memory for other reads and writes.

Append-only log structures are write-friendly, placing new data items at the end of a linear data structure called a "log" (Figure 4 (b)). To update an item, it simply inserts a new item to the log that overrides the previous value. Inserts and updates thus access memory sequentially, incurring fewer cache and TLB misses, making logs particularly suited for bulk data writes. This approach is common in flash memory stores due to the high cost of random flash writes [3, 4, 28], but has been used in only a few in-memory key-value systems [37].

Garbage collection is crucial to space efficiency. It reclaims space occupied by overwritten and deleted objects by moving live objects to a new log and removing the old log. Unfortunately, garbage collection is costly and often reduces performance because of the large amount of data it must copy, trading memory efficiency against request processing speed.

MICA's memory allocator: MICA uses separate memory allocators for cache and store semantics. Its cache mode uses a log structure with inexpensive garbage collection and in-place update support (Section 4.3.1). MICA's allocator provides fast inserts and updates, and exploits cache semantics to eliminate log garbage collection and drastically simplify free space defragmentation. Its store mode uses segregated fits [42, 47] that share the unified memory space to avoid rebalancing size classes (Section 4.3.3).

3.3.2 Indexing: Read-oriented vs. Write-friendly

Read-oriented index: Common choices for indexing are hash tables [15, 33, 36] or tree-like structures [30]. However, conventional data structures are much slower for writes compared to reads; hash tables examine many slots to find a space for the new item [15], and trees may require multiple operations to maintain structural invariants [30].

Write-friendly index: Hash tables using *chaining* [33, 36] can insert new items without accessing many memory locations, but they suffer a time-space tradeoff: by having long chains (few hash buckets), an item lookup must follow a long chain of items, this requiring multiple random dependent memory accesses; when chains are short (many hash buckets), memory overhead to store chaining pointers increases. *Lossy data structures* are rather unusual in in-memory key-value storage and studied only in limited contexts [7], but it is the standard design in hardware indexes such as CPU caches [21].

MICA's index: MICA uses new index data structures to offer both high-speed read and write. In cache mode, MICA's lossy index also leverages the cache semantics to achieve high insertion speed; it evicts an old item in the hash table when a hash collision occurs instead of spending system resources to resolve the collision. By using the memory allocator's eviction support, the MICA lossy index can avoid evicting recently-used items (Section 4.3.2). The MICA lossless index uses *bulk chaining*, which allocates cache line-aligned space to a bucket for each chain segment. This keeps the chain length short and space efficiency high (Section 4.3.3).

[2]Binary prefixes (powers of 2) end with an "i" suffix, whereas SI prefixes (powers of 10) have no "i" suffix.

4 MICA Design

This section describes each component in MICA and discusses how they operate together to achieve its goals.

4.1 Parallel Data Access

This section explains how CPU cores access data in MICA, but assumes that cores process only the requests for which they are responsible. Later in Section 4.2, we discuss how MICA assigns remote requests to CPU cores.

4.1.1 Keyhash-Based Partitioning

MICA creates one or more partitions per CPU core and stores key-value items in a partition determined by their key. Such horizontal partitioning is often used to shard *across* nodes [4, 11], but some key-value storage systems also use it across cores within a node [6, 25, 33].

MICA uses a *keyhash* to determine each item's partition. A keyhash is the 64-bit hash of an item's key calculated by the client and used throughout key-value processing in MICA. MICA uses the first few high order bits of the keyhash to obtain the partition index for the item.

Keyhash partitioning uniformly maps keys to partitions, reducing the request distribution imbalance. For example, in a Zipf-distributed population of size 192×2^{20} (192 Mi) with skewness 0.99 as used by YCSB [9],[3] the most popular key is 9.3×10^6 times more frequently accessed than the average; after partitioning keys into 16 partitions, however, the most popular partition is only 53% more frequently requested than the average.

MICA retains high throughput under this remaining partition-level skew because it can process requests in "hot" partitions more efficiently, for two reasons. First, a partition is popular *because* it contains "hot" items; these hot items naturally create locality in data access. With high locality, MICA experiences fewer CPU cache misses when accessing items. Second, the skew causes packet I/O to be more efficient for popular partitions (described in Section 4.2.1). As a result, throughput for the Zipf-distributed workload is 86% of the uniformly-distributed workload, making MICA's partitioned design practical even under skewed workloads.

4.1.2 Operation Modes

MICA can operate in EREW (Exclusive Read Exclusive Write) or CREW (Concurrent Read Exclusive Write). *EREW* assigns a single CPU core to each partition for all operations. No concurrent access to partitions eliminates synchronization and inter-core communication, making MICA scale linearly with CPU cores. *CREW* allows any core to read partitions, but only a single core can write. This combines the benefit of concurrent read and exclusive write; the former allows all cores to process read requests, while the latter still reduces expensive cache line transfer. CREW handles reads efficiently under highly skewed load, at the cost of managing read-write conflicts. MICA minimizes the synchronization cost with efficient optimistic locking [48] (Section 4.3.2).

Supporting cache semantics in CREW, however, raises a challenge for read (`GET`) requests: During a `GET`, the cache may need to update cache management information. For example, policies such as LRU use bookkeeping to remember recently used items, which can cause conflicts and cache-line bouncing among cores. This, in turn, defeats the purpose of using exclusive writes.

To address this problem, we choose an approximate approach: MICA counts reads only from the exclusive-write core. Clients round-robin CREW reads across all cores in a NUMA domain, so this is effectively a sampling-based approximation to, e.g., LRU replacement as used in MICA's item eviction support (Section 4.3.1).

To show performance benefits of EREW and CREW, our MICA prototype also provides the CRCW (Concurrent Read Concurrent Write) mode, in which MICA allows multiple cores to read and write any partition. This effectively models concurrent access to the shared data in non-partitioned key-value systems.

4.2 Network Stack

The network stack in MICA provides *network I/O* to transfer packet data between NICs and the server software, and *request direction* to route requests to an appropriate CPU core to make subsequent key-value processing efficient.

Exploiting the small key-value items that MICA targets, request and response packets use UDP. Despite clients not benefiting from TCP's packet loss recovery and flow/congestion control, UDP has been used widely for read requests (e.g., `GET`) in large-scale deployments of in-memory key-value storage systems [36] for low latency and low overhead. Our protocol includes sequence numbers in packets, and our application relies on the idempotency of `GET` and `PUT` operations for simple and stateless application-driven loss recovery, if needed: some queries may not be useful past a deadline, and in many cases, the network is provisioned well, making retransmission rare and congestion control less crucial [36].

4.2.1 Direct NIC Access

MICA uses Intel's DPDK [22] instead of standard socket I/O. This allows our user-level server software to control NICs and transfer packet data with minimal overhead. MICA differs from general network processing [12, 19, 41] that has used direct NIC access in that MICA is an application that processes high-level key-value requests.

In NUMA (non-uniform memory access) systems with multiple CPUs, NICs may have different affinities to CPUs. For example, our evaluation hardware has two

[3] i-th key constitutes $1/(i^{0.99} H_{n,0.99})$ of total requests, where $H_{n,0.99} = \sum_{i=1}^{n} (1/i^{0.99})$ and n is the total number of keys.

CPUs, each connected to two NICs via a direct PCIe bus. MICA uses NUMA-aware memory allocation so that each CPU and NIC only accesses packet buffers stored in their respective NUMA domains.

MICA uses NIC multi-queue support to allocate a dedicated RX and TX queue to each core. Cores exclusively access their own queues without synchronization in a similar way to EREW data access. By directing a packet to an RX queue, the packet can be processed by a specific core, as we discuss in Section 4.2.2.

Burst packet I/O: MICA uses the DPDK's burst packet I/O to transfer multiple packets (up to 32 in our implementation) each time it requests packets from RX queues or transmits them to TX queues. Burst I/O reduces the per-packet cost of accessing and modifying the queue, while adding only trivial delay to request processing because the burst size is small compared to the packet processing rate.

Importantly, burst I/O helps handle skewed workloads. A core processing popular partitions spends more time processing requests, and therefore performs packet I/O less frequently. The lower I/O frequency increases the burst size, reducing the per-packet I/O cost (Section 5.2). Therefore, popular partitions have more CPU available for key-value processing. An unpopular partition's core has higher per-packet I/O cost, but handles fewer requests.

Zero-copy processing: MICA avoids packet data copy throughout RX/TX and request processing. MICA uses MTU-sized packet buffers for RX even if incoming requests are small. Upon receiving a request, MICA avoids memory allocation and copying by reusing the request packet to construct a response: it flips the source and destination addresses and ports in the header and updates only the part of the packet payload that differs between the request and response.

4.2.2 Client-Assisted Hardware Request Direction

Modern NICs help scale packet processing by directing packets to different RX queues using hardware features such as Receiver-Side Scaling (RSS) and Flow Director (FDir) [12, 19, 41] based on the packet header.

Because each MICA key-value request is an individual packet, we wish to use *hardware* packet direction to directly send packets to the appropriate queue based upon the key. Doing so is much more efficient than redirecting packets in software. Unfortunately, the NIC alone cannot provide key-based request direction: RSS and FDir cannot classify based on the packet payload, and cannot examine variable length fields such as request keys.

Client assistance: We instead take advantage of the opportunity to co-design the client and server. The client caches information from a server directory about the operation mode (EREW or CREW), number of cores, NUMA domains, and NICs, and number of partitions.

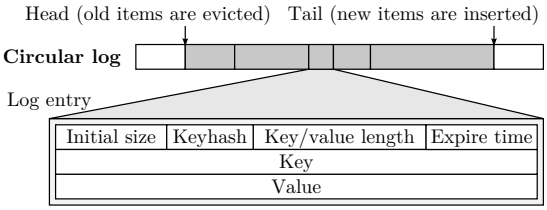


Figure 5: Design of a circular log.

The client then embeds the request direction information in the packet header: If the request uses exclusive data access (read/write on EREW and write on CREW), the client calculates the partition index from the keyhash of the request. If the request can be handled by any core (a CREW read), it picks a server core index in a round-robin way (across requests, but in the same NUMA domain (Section 4.2.1)). Finally, the client encodes the partition or core index as the UDP destination port.[4] The server programs FDir to use the UDP destination port, without hashing, ("perfect match filter" [23]), as an index into a table mapping UDP port numbers to a destination RX queue. Key hashing only slightly burdens clients. Using fast string hash functions such as CityHash [8], a single client machine equipped with dual 6-core CPUs on our testbed can generate over 40 M requests/second with client-side key hashing. Clients include the keyhash in requests, and servers reuse the embedded keyhash when they need a keyhash during the request processing to benefit from offloaded hash computation.

Client-assisted request direction using NIC hardware allows efficient request processing. Our results in Section 5.5 show that an optimized software-based request direction that receives packets from any core and distributes them to appropriate cores is significantly slower than MICA's hardware-based approach.

4.3 Data Structure

MICA, in cache mode, uses *circular logs* to manage memory for key-value items and *lossy concurrent hash indexes* to index the stored items. Both data structures exploit cache semantics to provide fast writes and simple memory management. Each MICA partition consists of a single circular log and lossy concurrent hash index.

MICA provides a store mode with straightforward extensions using segregated fits to allocate memory for key-value items and *bulk chaining* to convert the lossy concurrent hash indexes into lossless ones.

4.3.1 Circular Log

MICA stores items in its circular log by appending them to the *tail* of the log (Figure 5). This results in a space-efficient packing. It updates items in-place as long as the

[4] To avoid confusion between partition indices and the core indices, we use different ranges of UDP ports; a partition may be mapped to a core whose index differs from the partition index.

new size of the key+value does not exceed the size of the item when it was first inserted. The size of the circular log is bounded and does not change, so to add a new item to a full log, MICA evicts the oldest item(s) at the *head* of the log to make space.

Each entry includes the key and value length, key, and value. To locate the next item in the log and support item resizing, the entry contains the initial item size, and for fast lookup, it stores the keyhash of the item. The entry has an expire time set by the client to ignore stale data.

Garbage collection and defragmentation: The circular log eliminates the expensive garbage collection and free space defragmentation that are required in conventional log structures and dynamic memory allocators. Previously deleted items in the log are automatically collected and removed when new items enter the log. Almost all free space remains contiguously between the tail and head.

Exploiting the eviction of live items: Items evicted at the head are not reinserted to the log even if they have not yet expired. In other words, the log may delete items without clients knowing it. This behavior is valid in cache workloads; a key-value store must evict items when it becomes full. For example, Memcached [32] uses LRU to remove items and reserve space for new items.

MICA uses this item eviction to implement common eviction schemes at low cost. Its "natural" eviction is FIFO. MICA can provide LRU by reinserting any requested items at the tail because only the least recently used items are evicted at the head. MICA can approximate LRU by reinserting requested items selectively—by ignoring items recently (re)inserted and close to the tail; this approximation offers eviction similar to LRU without frequent reinserts, because recently accessed items remain close to the tail and far from the head.

A second challenge for conventional logs is that any reference to an evicted item becomes dangling. MICA does not store back pointers in the log entry to discover all references to the entry; instead, it provides detection, and removes dangling pointers incrementally (Section 4.3.2).

Low-level memory management: MICA uses hugepages and NUMA-aware allocation. Hugepages (2 MiB in x86-64) use fewer TLB entries for the same amount of memory, which significantly reduces TLB misses during request processing. Like the network stack, MICA allocates memory for circular logs such that cores access only local memory.

Without explicit range checking, accessing an entry near the end of the log (e.g., at $2^{34} - 8$ in the example below) could cause an invalid read or segmentation fault by reading off the end of the range. To avoid such errors without range checking, MICA manually maps the virtual memory addresses right after the end of the log to the same physical page as the first page of the log, making the entire log appear locally contiguous:

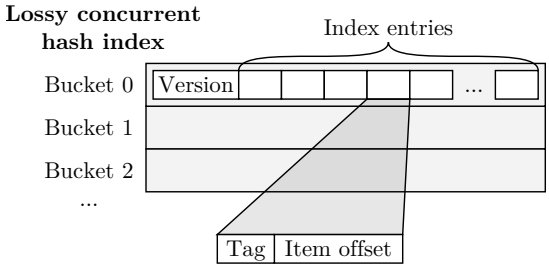
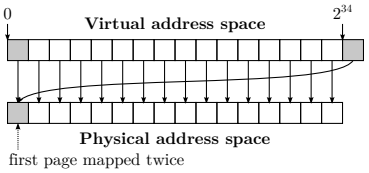


Figure 6: Design of a lossy concurrent hash index.



Our MICA prototype implements this scheme in userspace by mapping a pool of hugepages to virtual addresses using the `mmap()` system call.

4.3.2 Lossy Concurrent Hash Index

MICA's hash index locates key-value items in the log using a set-associative cache similar to that used in CPU caches. As shown in Figure 6, a hash index consists of multiple buckets (configurable for the workload), and each bucket has a fixed number of index entries (configurable in the source code; 15 in our prototype to occupy exactly two cache lines). MICA uses a portion of the keyhashes to determine an item's bucket; the item can occupy any index entry of the bucket unless there is a duplicate.

Each index entry contains partial information for the item: a tag and the item offset within the log. A tag is another portion of the indexed item's keyhash used for filtering lookup keys that do not match: it can tell whether the indexed item will never match against the lookup key by comparing the stored tag and the tag from the lookup keyhash. We avoid using a zero tag value by making it one because we use the zero value to indicate an empty index entry. Items are deleted by writing zero values to the index entry; the entry in the log will be automatically garbage collected.

Note that the parts of keyhashes used for the partition index, the bucket number, and the tag do not overlap. Our prototype uses 64-bit keyhashes to provide sufficient bits.

Lossiness: The hash index is lossy. When indexing a new key-value item into a full bucket of the hash index, the index evicts an index entry to accommodate the new item. The item evicted is determined by its age; if the item offset is most behind the tail of the log, the item is the oldest (or least recently used if the log is using LRU), and the associated index entry of the item is reclaimed.

This lossy property allows fast insertion. It avoids expensive resolution of hash collisions that lossless indexes of other key-value stores require [15, 33]. As a result,

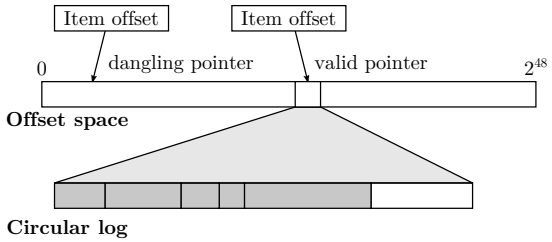


Figure 7: Offset space for dangling pointer detection.

MICA's insert speed is comparable to lookup speed.

Handling dangling pointers: When an item is evicted from the log, MICA does not delete its index entry. Although it is possible to store back pointers in the log entry, updating the hash index requires a random memory write and is complicated due to locking if the index is being accessed concurrently, so MICA does not. As a result, index pointers can "dangle," pointing to invalid entries.

To address this problem, MICA uses large pointers for head/tail and item offsets. As depicted in Figure 7, MICA's index stores log offsets that are wider than needed to address the full size of the log (e.g., 48-bit offsets vs 34 bits for a 16 GiB log). MICA detects a dangling pointer before using it by checking if the difference between the log tail and the item offset is larger than the actual log size.[5] If the tail wraps around the 48-bit size, however, a dangling pointer may appear valid again, so MICA scans the index incrementally to remove stale pointers.

This scanning must merely complete a full cycle before the tail wraps around in its wide offset space. The speed at which it wraps is determined by the increment rate of the tail and the width of the item offset. In practice, full scanning is infrequent even if writes occur very frequently. For example, with 48-bit offsets and writes occurring at 2^{30} bytes/second (millions of operations/second), the tail wraps every 2^{48-30} seconds. If the index has 2^{24} buckets, MICA must scan only 2^6 buckets per second, which adds negligible overhead.

Supporting concurrent access: MICA's hash index must behave correctly if the system permits concurrent operations (e.g., CREW). For this, each bucket contains a 32-bit version number. It performs reads optimistically using this version counter to avoid generating memory writes while satisfying `GET` requests [15, 30, 48]. When accessing an item, MICA checks if the initial state of the version number of the bucket is even-numbered, and upon completion of data fetch from the index and log, it reads the version number again to check if the final version number is equal to the initial version number. If either check fails, it repeats the read request processing from the beginning. For writes, MICA increments the version number by one before beginning, and increments the version number by one again after finishing all writes. In CRCW mode, which allows multiple writers to access the same bucket, a writer also spins until the initial version number is even (i.e., no other writers to this bucket) using a compare-swap operation instruction.

[5] $(\text{Tail} - \text{ItemOffset} + 2^{48}) \bmod 2^{48} > \text{LogSize}$.

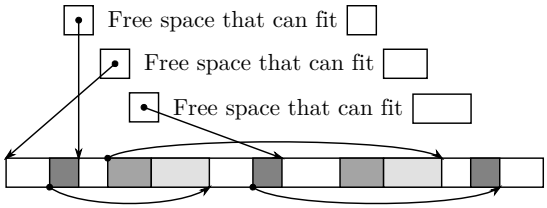


Figure 8: Segregated free lists for a unified space.

Our MICA prototype uses different code to optimize locking. It uses conventional instructions to manipulate version numbers to exploit memory access ordering on the x86 architecture [48] in CREW mode where there is only one writer. EREW mode does not require synchronization between cores, so MICA ignores version numbers. Because of such a hard-coded optimization, the current prototype lacks support for runtime switching between the operation modes.

Multi-stage prefetching: To retrieve or update an item, MICA must perform request parsing, hash index lookup, and log entry retrieval. These stages cause random memory access that can significantly lower system performance if cores stall due to CPU cache and TLB misses.

MICA uses multi-stage prefetching to interleave computation and memory access. MICA applies memory prefetching for random memory access done at each processing stage in sequence. For example, when a burst of 8 RX packets arrives, MICA fetches packets 0 and 1 and *prefetches* packets 2 and 3. It decodes the requests in packets 0 and 1, and prefetches buckets of the hash index that these requests will access. MICA continues packet payload prefetching for packets 4 and 5. It then prefetches log entries that may be accessed by the requests of packets 0 and 1 while prefetching the hash index buckets for packets 2 and 3, and the payload of packet 6 and 7. MICA continues this pipeline until all requests are processed.

4.3.3 Store Mode

The store mode of MICA uses segregated fits [42, 47] similar to fast malloc implementations [27], instead of the circular log. Figure 8 depicts this approach. MICA defines multiple size classes incrementing by 8 bytes covering all supported item sizes, and maintains a freelist for each size class (a linked list of pointers referencing unoccupied memory regions that are at least as large as the size class). When a new item is inserted, MICA chooses the smallest size class that is at least as large as the item size and has any free space. It stores the item in the free space, and inserts any unused region of the free space into a freelist that matches that region's size. When an item is deleted, MICA coalesces any adjacent free regions using boundary

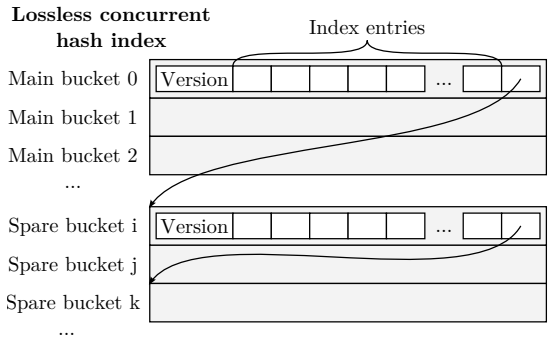


Figure 9: Bulk chaining in MICA's lossless hash index.

tags [26] to recreate a large free region.

MICA's segregated fits differ from the simple segregated storage used in Memcached [15, 32]. MICA maintains a unified space for all size classes; on the contrary, Memcached's SLAB allocator dynamically assigns memory blocks to size classes, which effectively partitions the memory space according to size classes. The unified space of MICA eliminates the need to rebalance size classes unlike the simple segregated storage. Using segregated fits also makes better use of memory because MICA already has partitioning done with keyhashes; a SLAB allocator introducing another partitioning would likely waste memory by allocating a whole block for only a few items, resulting in low memory occupancy.

MICA converts its lossy concurrent hash index into a lossless hash index by using *bulk chaining*. Bulk chaining is similar to the traditional chaining method in hash tables; it adds more memory space to the buckets that contain an excessive number of items.

Figure 9 shows the design of the lossless hash index. MICA uses the lossy concurrent hash index as the main buckets and allocates space for separate spare buckets that are fewer than the main buckets. When a bucket experiences an overflow, whether it is a main bucket or spare bucket, MICA adds an unused spare bucket to the full bucket to form a bucket chain. If there are no more spare buckets available, MICA rejects the new item and returns an out-of-space error to the client.

This data structure is friendly to memory access. The main buckets store most of items (about 95%), keeping the number of random memory read for an index lookup close to 1; as a comparison, cuckoo hashing [39] used in improved Memcached systems [15] would require 1.5 random memory accesses per index lookup in expectation. MICA also allows good memory efficiency; because the spare buckets only store overflow items, making the number of spare buckets 10% of the main buckets allows the system to store the entire dataset of 192 Mi items in our experiments (Section 5).

5 Evaluation

We answer four questions about MICA in this section:

- *Does it perform well under diverse workloads?*
- *Does it provide good latency?*
- *How does it scale with more cores and NIC ports?*
- *How does each component affect performance?*

Our results show that MICA has consistently high throughput and low latency under a variety of workloads. It scales nearly linearly, using CPU cores and NIC ports efficiently. Each component of MICA is needed. MICA achieves 65.6–76.9 million operations/second (Mops), which is over 4–13.5x faster than the next fastest system; the gap widens as the fraction of write requests increases.

MICA is written in 12 K lines of C and runs on x86-64 GNU/Linux. Packet I/O uses the Intel DPDK 1.4.1 [22].

Compared systems: We use custom versions of open-source Memcached [32], MemC3 [15], Masstree [30], and RAMCloud [37]. The revisions of the original code we used are: Memcached: 87e2f36; MemC3: an internal version; Masstree: 4ffb946; RAMCloud: a0f6889.

Note that the compared systems often offer additional capabilities compared to others. For example, Masstree can handle range queries, and RAMCloud offers low latency processing on InfiniBand; on the other hand, these key-value stores do not support automatic item eviction as Memcached systems do. Our evaluation focuses on the performance of the standard features (e.g., single key queries) common to all the compared systems, rather than highlighting the potential performance impact from these semantic differences.

Modifications to compared systems: We modify the compared systems to use our lightweight network stack to avoid using expensive socket I/O or special hardware (e.g., InfiniBand). When measuring Memcached's baseline latency, we use its original network stack using the kernel to obtain the latency distribution that typical Memcached deployments would experience. Our experiments do not use any client-side request batching. We also modified these systems to invoke memory allocation functions though our framework if they use hugepages, because the DPDK requests all hugepages from the OS at initialization and would make the unmodified systems inoperable if they request hugepages from the OS; we kept other memory allocations using no hugepages as-is. Finally, while running experiments, we found that statistics collection in RAMCloud caused lock contention, so we disabled it for better multi-core performance.

5.1 Evaluation Setup

Server/client configuration: MICA server runs on a machine equipped with dual 8-core CPUs (Intel Xeon E5-2680 @2.70 GHz), 64 GiB of total system memory, and eight 10-Gb Ethernet ports (four Intel X520-T2's). Each

CPU has 20 MiB of L3 cache. We disabled logical processor support ("Hyper-Threading"). Each CPU accesses the 32 GiB of the system memory that resides in its local NUMA domain over a quad-channel DDR3-1600 bus. Each CPU socket is directly connected to two NICs using PCIe gen2. Access to hardware resources in the remote NUMA domain uses an interconnect between two CPUs (Intel QuickPath).

We reserved the half of the memory (16 GiB in each NUMA domain) for hugepages regardless of how MICA and the compared systems use hugepages.

MICA allocates 16 partitions in the server, and these partitions are assigned to different cores. We configured the cache version of MICA to use approximate LRU to evict items; MICA reinserts any recently accessed item at the tail if the item is closer to the head than to the tail of the circular log.

Two client machines with dual 6-core CPUs (Intel Xeon L5640 @2.27 GHz) and two Intel X520-T2's generate workloads. The server and clients are directly connected without a switch. Each client is connected to the NICs from both NUMA domains of the server, allowing a client to send a request to any server CPU.

Workloads: We explore different aspects of the systems by varying the item size, skew, and read-write ratio.

We use three datasets as shown in the following table:

Dataset	Key Size (B)	Value Size (B)	Count
Tiny	8	8	192 Mi
Small	16	64	128 Mi
Large	128	1024	8 Mi

We use two workload types: *uniform* and *skewed*. Uniform workloads use the same key popularity for all requests; skewed workloads use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same as YCSB's [9].

Workloads have a *varied ratio between* `GET` *and* `PUT`. 50% `GET` (50% `PUT`) workloads are write-intensive, and 95% `GET` (5% `PUT`) workloads are read-intensive. They correspond to YCSB's A and B workloads, respectively.

Workload generation: We use our custom key-value request generator that uses similar techniques to our lightweight network stack to send more than 40 Mops of key-value requests per machine to saturate the link.[6] It uses approximation techniques of Zipf distribution generation [17, 38] for fast skewed workload generation.

To find the maximum *meaningful* throughput of a system, we adjust the workload generation rate to allow only marginal packet losses ($< 1\%$ at any NIC port). We could generate requests at the highest rate to cause best-effort request processing (which can boost measured throughput more than 10%), as is commonly done in throughput measurement of software routers [12, 19], but we avoid this method because we expect that real deployments of in-memory key-value stores would not tolerate excessive packet losses, and such flooding can distort the intended skew in the workload by causing biased packet losses at different cores.

The workload generator does not receive every response from the server. On our client machines, receiving packets whose size is not a multiple of 64 bytes is substantially slower due to an issue in the PCIe bus [18].

The workload generator works around this slow RX by sampling responses to perform fewer packet RX from NIC to CPU. It uses its real source MAC addresses for only a fraction of requests, causing its NIC to drop the responses to the other requests. By looking at the sampled responses, the workload generator can validate that the server has correctly processed the requests. Our server is unaffected from this issue and performs full packet RX.

5.2 System Throughput

We first compare the full-system throughput. MICA uses EREW with all 16 cores. However, we use a different number of cores for the other systems to obtain their best throughput because some of them (Memcached, MemC3, and RAMCloud) achieve higher throughput with fewer cores (Section 5.4). The throughput numbers are calculated from the actual number of responses sent to the clients after processing the requests at the server. We denote the cache version of MICA by MICA-c and the store version of MICA by MICA-s.

Figure 10 (top) plots the experiment result using *tiny key-value items*. MICA performs best, regardless of the skew or the `GET` ratio. MICA's throughput reaches 75.5–76.9 Mops for uniform workloads and 65.6–70.5 Mops for skewed ones; its parallel data access does not incur more than a 14% penalty for skewed workloads. MICA uses 54.9–66.4 Gbps of network bandwidth at this processing speed—this speed is very close to 66.6 Gbps that our network stack can handle when doing packet I/O only. The next best system is Masstree at 16.5 Mops, while others are below 6.1 Mops. All systems except MICA suffer noticeably under write-intensive 50% `GET`.

Small key-value items show similar results in Figure 10 (middle). However, the gap between MICA and the other systems shrinks because MICA becomes network bottlenecked while the other systems never saturate the network bandwidth in our experiments.

Large key-value items, shown in Figure 10 (bottom), exacerbates the network bandwidth bottleneck, further limiting MICA's throughput. MICA achieves 12.6–14.6 Mops for 50% `GET` and 8.6–9.4 Mops for 95% `GET`; note that MICA shows high throughput with lower `GET` ratios,

[6] MICA clients are still allowed to use standard socket I/O in cases where the socket overhead on the client machines is acceptable because the MICA server and clients use the plain UDP protocol.

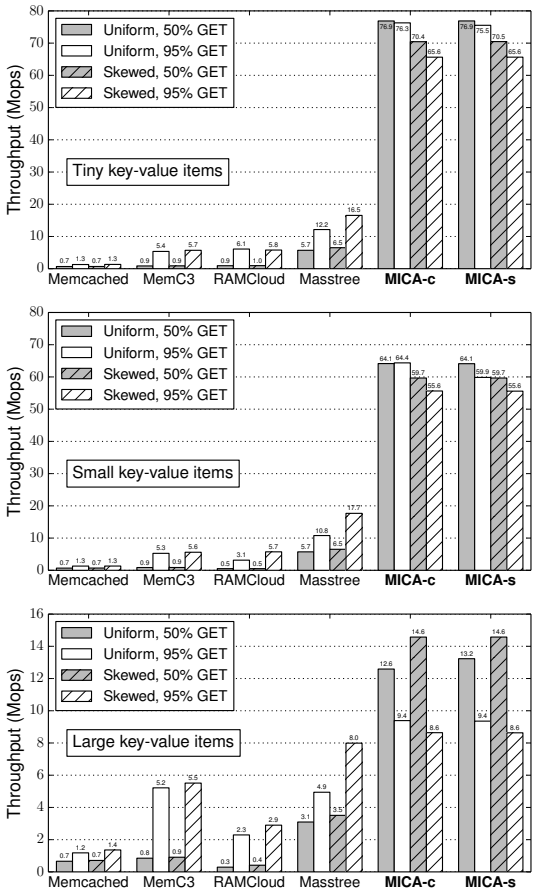


Figure 10: End-to-end throughput of in-memory key-value systems. All systems use our lightweight network stack that does not require request batching. The bottom graph (large key-value items) uses a different Y scale from the first two graphs'.

which require less network bandwidth as the server can omit the key and value from the responses. Unlike MICA, however, all other systems achieve higher throughput under 95% `GET` than under 50% `GET` because these systems are bottleneck locally, not by the network bandwidth.

In those measurements, MICA's cache and store modes show only minor differences in the performance. We will refer to the cache version of MICA as MICA in the rest of the evaluation for simplicity.

Skew resistance: Figure 11 compares the per-core throughput under uniform and skewed workloads of 50% `GET` with tiny items. MICA uses EREW. Several cores process more requests under the skewed workload than under the uniform workload because they process requests more efficiently. The skew in the workload increases the RX burst size of the most loaded core from 10.2 packets per I/O to 17.3 packets per I/O, reducing its per-packet I/O cost, and the higher data locality caused by the workload skew improves the average cache hit ratio of all cores from

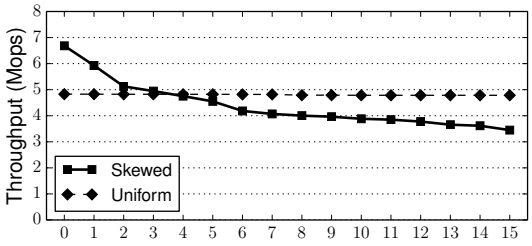


Figure 11: Per-core breakdown of end-to-end throughput.

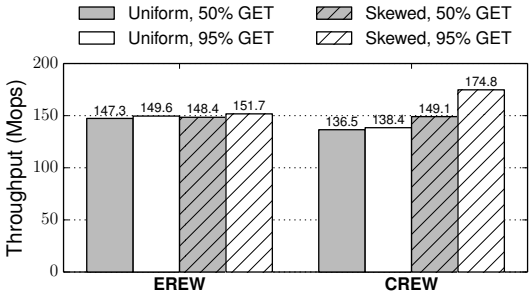


Figure 12: Local throughput of key-value data structures.

67.8% to 77.8%. A local benchmark in Figure 12 (without network processing) also shows that skewed workloads grant good throughput for local key-value processing due to the data locality. These results further justify the partitioned design of MICA and explains why MICA retains high throughput under skewed workloads.

Summary: MICA's throughput reaches 76.9 Mops, at least 4x faster than the next best system. MICA delivers consistent performance across different skewness, write-intensiveness, and key-value sizes.

5.3 Latency

To show that MICA achieves comparably low latency while providing high throughput, we compare MICA's latency with that of the original Memcached implementation that uses the kernel network stack. To measure the end-to-end latency, clients tag each request packet with the current timestamp. When receiving responses, clients compare the current timestamp and the previous timestamp echoed back in the responses. We use uniform 50%

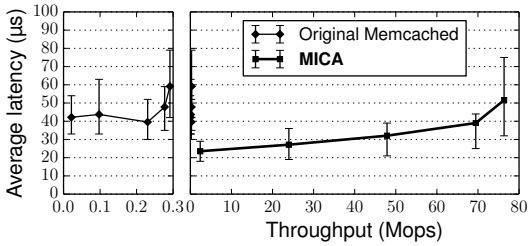


Figure 13: End-to-end latency of the original Memcached and MICA as a function of throughput.

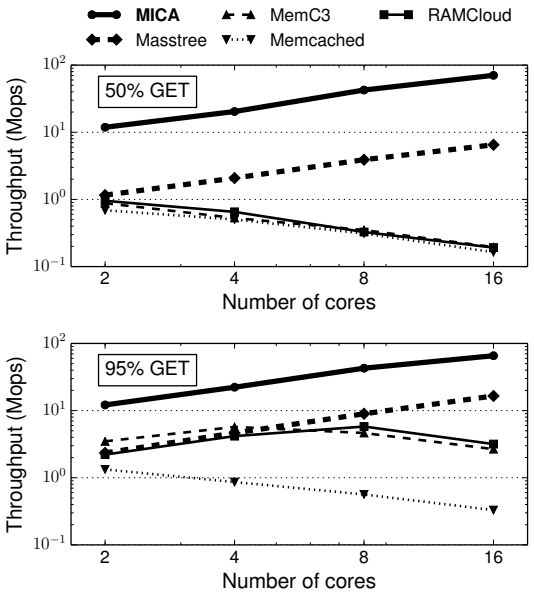


Figure 14: End-to-end throughput of in-memory key-value systems using a varying number of cores. All systems use our lightweight network stack.

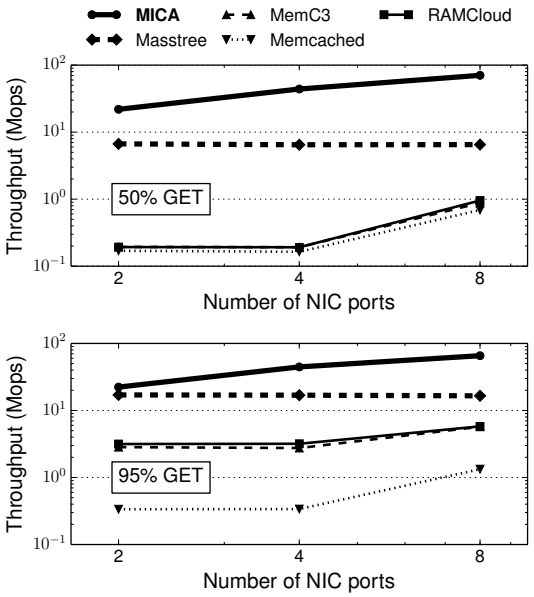


Figure 15: End-to-end throughput of in-memory key-value systems using a varying number of NIC ports. All systems use our lightweight network stack.

GET workloads on tiny items. MICA uses EREW. The client varies the request rate to observe the relationship between throughput and latency.

Figure 13 plots the end-to-end latency as a function of throughput; the error bars indicate 5th- and 95th-percentile latency. The original Memcached exhibits almost flat latency up to certain throughput, whereas MICA shows varied latency depending on the throughput it serves. MICA's latency lies between 24–52 µs. At the similar latency level of 40 µs, MICA shows 69 Mops—more than two orders of magnitude faster than Memcached.

Because MICA uses a single round-trip per request unlike RDMA-based systems [35], we believe that MICA provides best-in-class low-latency key-value operations.

Summary: MICA achieves both high throughput and latency near the network minimum.

5.4 Scalability

CPU scalability: We vary now the number of CPU cores and compare the end-to-end throughput. We allocate cores evenly to both NUMA domains so that cores can efficiently access NICs connected to their CPU socket. We use skewed workloads on tiny items because it is generally more difficult for partitioned stores to handle skewed workloads. MICA uses EREW.

Figure 14 (upper) compares core scalability of systems with 50% GET. Only MICA and Masstree perform better with more cores. Memcached, MemC3, and RAMCloud scale poorly, achieving their best throughput at 2 cores.

The trend continues for 95% GET requests in Figure 14 (lower); MICA and Masstree scale well as before. The rest also achieve higher throughput, but still do not scale. Note that some systems scale differently from their original papers. For example, MemC3 achieves 5.7 Mops at 4 cores, while the original paper shows 4.4 Mops at 16 cores [15]. This is because using our network stack instead of their network stack reduces I/O cost, which may expose a different bottleneck (e.g., key-value data structures) that can change the optimal number of cores for the best throughput.

Network scalability: We also change the available network bandwidth by varying the number of NIC ports we use for request processing. Figure 15 shows that MICA again scales well with high network bandwidth, because MICA can use almost all available network bandwidth for request processing. The GET ratio does not affect the result for MICA significantly. This result suggests that MICA can possibly scale further with higher network bandwidth (e.g., multiple 40 Gbps NICs). MICA and Masstree achieve similar performance under the 95% GET workload when using 2 ports, but Masstree and other systems do not scale well with more ports.

Summary: MICA scales well with more CPU cores and more network bandwidth, even under write-intensive workloads where other systems tend to scale worse.

5.5 Necessity of the Holistic Approach

In this section, we demonstrate how each component of MICA contributes to its performance. Because MICA is a coherent system that exploits the synergy between its

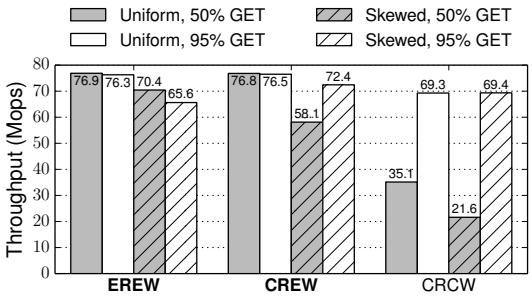


Figure 16: End-to-end performance using MICA's EREW, CREW, and CRCW.

Method	Workload	Throughput
Software-only	Uniform	33.9 Mops
	Skewed	28.1 Mops
Client-assisted hardware-based	Uniform	76.9 Mops
	Skewed	70.4 Mops

Table 1: End-to-end throughput of different request direction methods.

components, we compare different approaches for one component while keeping the other components the same.

Parallel data access: We use end-to-end experiments to measure how different data access modes affect the system performance. We use tiny items only. Figure 16 shows the end-to-end results. EREW shows consistently good performance. CREW achieves slightly higher throughput with high `GET` ratios on skewed workloads compared to EREW (white bars at 95% `GET`) because despite the overheads from bucket version management, CREW can use multiple cores to read popular items without incurring excessive inter-core communication. While CRCW performs better than any other compared systems (Section 5.2), CRCW offers no benefit over EREW and CREW; this suggests that we should avoid CRCW.

Network stack: As shown in Section 5.2, switching Masstree to our network stack resulted in much higher throughput (16.5 Mops without request batching) than the throughput from the original paper (8.9 Mops with request batching [30]); this indicates that our network stack provides efficient I/O for key-value processing.

The next question is how important it is to use hardware to direct requests for exclusive access in MICA. To compare with MICA's client-assisted hardware request direction, we implemented software-only request direction: clients send requests to any server core in a round-robin way, and the server cores direct the received requests to the appropriate cores for EREW data access. We use Intel DPDK's queue to implement message queues between cores. We use 50% `GET` on tiny items.

Table 1 shows that software request direction achieves only 40.0–44.1% of MICA's throughput. This is due to the inter-core communication overhead of software request direction. Thus, MICA's request direction is crucial for realizing the benefit of exclusive access.

Method	Workload	Throughput
Partitioned Masstree	50% `GET`	5.8 Mops
	95% `GET`	17.9 Mops
MICA	50% `GET`	70.4 Mops
	95% `GET`	65.6 Mops

Table 2: End-to-end throughput comparison between partitioned Masstree and MICA using skewed workloads.

Key-value data structures: MICA's circular logs, lossy concurrent hash indexes, and bulk chaining permit high-speed read and write operations with simple memory management. Even CRCW, the slowest data access mode of MICA, outperforms the second best system, Masstree (Section 5.2).

We also demonstrate that partitioning existing data structures does not simply grant MICA's high performance. For this, we compare MICA with "partitioned" Masstree, which uses one Masstree instance per core, with its support for concurrent access disabled in the source code. This is similar to MICA's EREW. We also use the same partitioning and request direction scheme.

Table 2 shows the result with skewed workloads on tiny items. Partitioned Masstree achieves only 8.2–27.3% of MICA's performance, with the throughput for 50% `GET` even lower than non-partitioned Masstree (Section 5.2). This indicates that to make best use of MICA's parallel data access and network stack, it is important to use key-value data structures that perform high-speed writes and to provide high efficiency with data partitioning.

In conclusion, the holistic approach is essential; any missing component significantly degrades performance.

6 Related Work

Most DRAM stores are not partitioned: Memcached [32], RAMCloud [37], MemC3 [15], Masstree [30], and Silo [45] all have a single partition for each server node. Masstree and Silo show that partitioning can be efficient under some workloads but is slow under workloads with a skewed key popularity and many cross-partition transactions. MICA exploits burst I/O and locality so that even in its exclusive EREW mode, loaded partitions run faster. It can do so because the simple key-value requests that it targets do not cross partitions.

Partitioned systems are fast with well-partitioned data. Memcached on Tilera [6], CPHash [33], and Chronos [25] are partitioned in-memory key-value systems that exclusively access partitioned hash tables to minimize lock contention and cache movement, similar to MICA's EREW partitions. These systems lack support for other partitioning such as MICA's CREW that can provide higher throughput under read-intensive skewed workloads.

H-Store [44] and VoltDB [46] use single-threaded execution engines that access their own partition exclusively, avoiding expensive concurrency control. Because workload skew can reduce system throughput, they require careful data partitioning, even using machine learning methods [40], and dynamic load balancing [25]. MICA achieves similar throughput under both uniform and skewed workloads without extensive partitioning and load balancing effort because MICA's keyhash-based partitioning mitigates the skew using and its request processing for popular partitions exploits burst packet I/O and cache-friendly memory access.

Several in-memory key-value systems focus on low latency request processing. RAMCloud achieves 4.9–15.3 µs end-to-end latency for small objects [1], and Chronos exhibits average latency of 10 µs and a 99th-percentile latency of 30 µs, on low latency networks such as InfiniBand and Myrinet. Pilaf [35] serves read requests using one-sided RDMA reads on a low-latency network. Our MICA prototype currently runs on 10-Gb Ethernet NIC whose base latency is much higher [16]; we plan to evaluate MICA on a low-latency network.

Prior work studies providing a high performance reliable transport service using low-level unreliable datagram services. The Memcached UDP protocol relies on application-level packet loss recovery [36]. Low-overhead user-level implementations for TCP such as mTCP [24] can offer reliable communication to Memcached applications without incurring high performance penalties. Low-latency networks such as InfiniBand often implement hardware-level reliable datagrams [35].

Affinity-Accept [41] uses Flow Director on the commodity NIC hardware to load balance TCP connections across multiple CPU cores. Chronos [25] directs remote requests to server cores using client-supplied information, similar to MICA; however, Chronos uses software-based packet classification whose throughput for small key-value requests is significantly lower than MICA's hardware-based classification.

Strict or complex item eviction schemes in key-value stores can be so costly that it can reduce system throughput significantly. MemC3 [15] replaces Memcached [32]'s original LRU with a CLOCK-based approximation to avoid contention caused by LRU list management. MICA's circular log and lossy concurrent hash index use its lossy property to support common eviction schemes at low cost; the lossy concurrent hash index is easily extended to support lossless operations by using bulk chaining.

A worthwhile area of future work is applying MICA's techniques to semantically richer systems, such as those that are durable [37], or provide range queries [13, 30] or multi-key transactions [45]. Our results show that existing systems such as Masstree can benefit considerably simply by moving to a lightweight network stack; nevertheless, operations in these systems may cross partitions, it remains to be seen how to best harness the speed of exclusively accessed partitions.

7 Conclusion

MICA is an in-memory key-value store that provides high-performance, scalable key-value storage. It provides consistently high throughput and low latency for read/write-intensive workloads with a uniform/skewed key popularity. We demonstrate high-speed request processing with MICA's parallel data access to partitioned data, efficient network stack that delivers remote requests to appropriate CPU cores, and new lossy and lossless data structures that exploit properties of key-value workloads to provide high-speed write operations without complicating memory management.

Acknowledgments

This work was supported by funding from the National Science Foundation under awards CCF-0964474 and CNS-1040801, Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and Basic Science Research Program through the National Research Foundation of Korea funded by MSIP (NRF-2013R1A1A1076024). Hyeontaek Lim was supported in part by the Facebook Fellowship. We would like to thank Nick Feamster, John Ousterhout, Dong Zhou, Yandong Mao, Wyatt Lloyd, and our NSDI reviewers for their valuable feedback, and Prabal Dutta for shepherding this paper.

References

[1] Ramcloud project wiki: clusterperf November 12, 2012, 2012. https://ramcloud.stanford.edu/wiki/display/ramcloud/clusterperf+November+12%2C+2012.

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, Feb. 2012.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, Apr. 2010.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[6] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store.

http://gigaom2.files.wordpress.com/2011/07/facebook-tilera-whitepaper.pdf, 2011.

[7] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, June 2013.

[8] CityHash. http://code.google.com/p/cityhash/, 2014.

[9] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[10] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[12] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[13] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. ACM SIGCOMM*, Aug. 2012.

[14] Facebook's memcached multiget hole: More machines != more capacity. http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html, 2009.

[15] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[16] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, June 2013.

[17] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, May 1994.

[18] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient support for evolvable internetworking. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[19] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[20] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, Oct. 2012.

[21] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12): 1612–1630, Dec. 1989.

[22] Intel. Intel Data Plane Development Kit (Intel DPDK). http://www.intel.com/go/dpdk, 2014.

[23] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, 2014.

[24] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[25] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, Oct. 2012.

[26] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997. First edition published in 1968.

[27] D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.

[28] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[29] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with Smart Pipes: Designing SoC accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, June 2013.

[30] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2012.

[31] Mellanox ConnectX-3 product brief. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf, 2013.

[32] A distributed memory object caching system. http://memcached.org/, 2014.

[33] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2012.

[34] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, July 2002.

[35] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 conference on USENIX Annual technical conference*, June 2013.

[36] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[37] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[38] Optimized approximative pow() in C / C++. http://martin.ankerl.com/2012/01/25/optimized-approximative-pow-in-c-and-cpp/, 2012.

[39] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[40] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD '12: Proceedings of the 2012 international conference on Management of Data*, May 2012.

[41] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, Apr. 2012.

[42] P. Purdom, S. Stigler, and T.-O. Cheam. Statistical investigation of three storage allocation algorithms. *BIT Numerical Mathematics*, 11(2), 1971.

[43] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012.

[44] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proc. VLDB*, Sept. 2007.

[45] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[46] VoltDB, the NewSQL database for high velocity applications. http://voltdb.com/, 2014.

[47] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 1995.

[48] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.