

Building Memory Efficient Key-Value Stores in the CXL Era

Theodore Michailidis
University of California, San Diego
San Diego, CA, USA

Steven Swanson
University of California, San Diego
San Diego, CA, USA

Jishen Zhao
University of California, San Diego
San Diego, CA, USA

Abstract

Key-value (KV) stores are utilized in a wide range of modern applications, with in-memory KV stores being popular for their low-latency and high throughput. In-memory KV stores, like redis and memcached, use hashtable-based indices that offer on average constant insert, lookup and delete times. However, these indices impose a high memory overhead and reduced memory and cache performance by design. While the index itself is compact in memory, accesses to data and metadata (e.g., key sizes, value sizes) lead to significant memory overhead. Moreover, traditional memory tiering mechanisms for heterogeneous memory architectures struggle to efficiently manage KV stores, leading to significant performance degradation.

To solve these issues, we introduce CBC (Channel-Billed Cuckoo), a keyless, compact, memory-efficient and tiering-aware key-value index. CBC uses a 2 MiB-backed cuckoo hashing scheme that significantly extends TLB reach and improves cache locality, while also reducing the index memory overhead. Furthermore, we introduce a new KV-specific memory tiering mechanism that improves memory tiering performance by reducing profiling overhead and avoiding unnecessary page migrations. Our results show that CBC minimizes TLB and cache pollution, while it achieves up to 62% end-to-end lower memory overhead and one-to-two orders of magnitude better throughput over redis. Last, our evaluation also shows that CBC's tiering mechanism outperforms previous memory tiering mechanisms by up to 30%.

CCS Concepts: • Software and its engineering → Memory management; • Hardware → Memory and dense storage; • Information systems → Key-value stores.

Keywords: Key-value stores, cuckoo hashing, memory tiering, CXL, heterogeneous memory, huge pages, TLB

ACM Reference Format:

Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2026. Building Memory Efficient Key-Value Stores in the CXL Era. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC*

2026), November 15-18, 2026, Hong Kong, China. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3828523.3857012>

1 Introduction

Key-value (KV) stores are widely used in modern applications, offering a simple data abstraction with a collection of key-value pairs. Most commercially-used KV stores, like redis and memcached, use hashtable-based indices. These indices provide average-case $O(1)$ time complexity for lookups, inserts, and deletions, making them highly efficient for workloads with frequent read and write operations. However, designing indices for key-value stores involves several trade-offs between memory efficiency, lookup performance, and implementation complexity. These indices incur significant memory overhead due to metadata, pointer storage, and collision resolution mechanisms such as chaining.

Another issue with KV stores is huge pages' negative effect. Huge pages are a memory management feature that allow the operating system to use larger memory page sizes (usually 2 MiB or 1 GiB) instead of the default 4 KiB. The use of huge pages reduces the number of page entries required for address translation, lowering the translation overhead and improving the Translation Lookaside Buffer (TLB) coverage. While their use is beneficial for applications with large memory footprints, commercial KV stores are advising against their use [6, 17, 22] primarily due to increased latency and memory bloating.

Recent studies have shown that DRAM is the most expensive component in a server, accounting for up to 50% of the total server cost [1, 29]. At the same time, Weiner et al. [29] showed that a large part of allocated DRAM is kept idle. These issues have led the research community to explore new hardware and software technologies to reduce memory idling and the cost of memory. Compute Express Link (CXL) is part of the hardware efforts to reduce the cost and use of DRAM. CXL is an industry standard that enables high-bandwidth, low-latency communication between CPUs, memory, and accelerators. Heterogeneous memory systems can exploit CXL to complement DRAM with slower and more affordable CXL-attached memories.

A popular software solution is memory tiering, an emerging paradigm that leverages a hierarchy of memory types to optimize performance and cost-efficiency. It involves using a combination of fast memory and a slower memory (e.g. CXL-attached memories) to store data according to their access patterns and latency. The goal is to keep frequently



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11

<https://doi.org/10.1145/3828523.3857012>

accessed data in the fastest memory tier while offloading less frequently accessed data to slower tiers. As in-memory databases like key-value stores grow in size, memory tiering becomes crucial for managing large datasets without incurring prohibitive costs.

However, we notice that current memory tiering techniques for key-value stores face significant challenges. **First**, existing key-value stores often use memory-inefficient data structures, leading to high memory overheads. This inefficiency is exacerbated when deploying in slower memory tiers, where access latencies are higher. **Second**, KV stores have mostly random access patterns, which leads to exacerbated performance degradation due to excessive memory profiling and migrations.

We introduce *Channel-Billed Cuckoo* (CBC)¹, a keyless, memory-efficient, high-performance key-value index, designed to be memory and cache efficient, while also being optimized for memory tiering. CBC uses a compact hash table design that reduces the memory overhead by eliminating the need to store full keys. Our design significantly reduces the memory footprint of the index, allowing CBC to store more entries in memory and improving cache locality. To the best of our knowledge, CBC is the first KV index that fully discards key storage, replacing them with a constant-size, key-equivalent set of hashes. CBC also uses large 2 MiB pages for the hash table, which improves TLB performance. To optimize for memory tiering, CBC employs a new mechanism that reduces memory profiling and migrations, improving the performance in heterogeneous memory systems.

The remainder of the paper is organized as follows. Section 2 provides the background and motivation for this work. Section 3 describes the design and Section 4 includes the implementation details of CBC. In Section 5 we present our evaluation of CBC against existing key-value stores and memory tiering mechanisms. Section 6 contains the related work and Section 7 concludes our paper.

2 Background and Motivation

This section contains information regarding the background of our work.

2.1 Existing KV Indices

Popular KVs, like redis and memcached, use as their index hash tables that contain multiple buckets where KV pairs are stored. Hash collisions are handled using chaining, where multiple key-value pairs that hash to the same bucket are stored in a linked list. To maintain performance, these hash tables are doubled in size when the number of elements surpasses a pre-existing threshold. For memcached this limit is 1.5 elements per bucket, while redis uses a different approach: when heavy background operations are

¹The channel-billed cuckoo is the largest cuckoo in existence, similarly to a huge-paged cuckoo hash table.

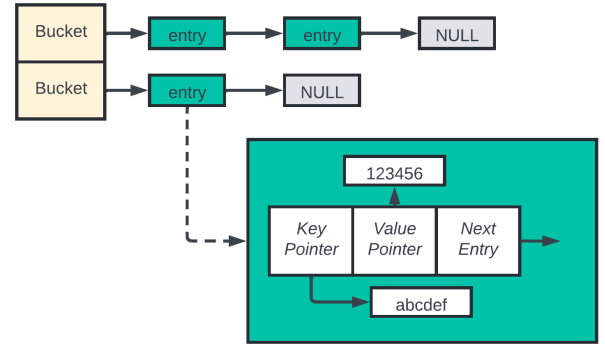


Figure 1. Hashtable view in redis and memcached.

absent, redis uses a resizing threshold of 1 element per bucket. However, when a heavy background operation is underway, e.g. persisting the KV, redis switches to a resizing threshold of 4 elements per bucket, to minimize performance degradation.

Figure 1 shows the hashtable overview in traditional KV stores like redis and memcached. Each entry contains at least 3×8-byte fields: the pointer to the key, the pointer to the value and the pointer to the next element in the linked list. This leads to an entry size of at least 24 bytes plus the bytes used to store the key and value.

For applications that use KVs for fast value retrieval, storing pointers to the keys is inefficient for 2 reasons: **i.** Accessing a key needs two memory accesses: one for the pointer to the key (entry), and one for the actual key. If the entry to look up is not the first in the bucket, then traversing the linked list requires more accesses. **ii.** Storing the whole key can be space-inefficient, especially as the size of the key increases. Currently, redis allows key sizes of up to 512 MB [21], while memcached has a maximum key size of 250 bytes.

2.2 Cuckoo Hashing

Cuckoo hashing is a collision resolution algorithm that allows elements to reside in multiple possible locations. Traditional cuckoo hashing (*2-ary*) involves two hash tables (T_1 and T_2) with two different hash functions (F_1 and F_2), one for each table. During insertion, an element x can be placed either in $T_1[F_1(x)]$ or $T_2[F_2(x)]$. If the target location is occupied, the residing element y is evicted and placed in the appropriate position in the other table. This process continues until the evicted element is placed in a free location. Lookups and deletions for x need to check all possible locations, namely $T_1[F_1(x)]$ and $T_2[F_2(x)]$.

Figure 2 shows how insertion works in a 2-ary cuckoo hash table. Initially, the hash table contains 4 elements a , b , c and d . In Step ①, we insert e at $T_1[F_1(e)]$, which kicks out d . In Step ② we insert d at $T_2[F_2(d)]$, which kicks out c .

Finally, in Step ③ we insert c at $T_1[F_1(c)]$, which is empty, thus the insertion process concludes.

2.3 Cuckoo Hashing Collisions

Cuckoo hash tables' major drawback lies in collisions during insertion. Insertions on a 2-ary cuckoo hash table with a load factor less than 50% [25, 32] are almost guaranteed to succeed, taking constant time with at most a few collisions. As the load factor increases above 50%, the insertion success rate drops dramatically, with collisions increasing and performance dropping significantly. Shortly after surpassing the 50% load limit, fresh insertions lead to never-ending unsuccessful re-insertions. A popular cuckoo hashing variation that solves this problem increases the number of hash tables from two to three, which allows for a load factor of up to 90% [32] before insertions start to fail.

Another popular variation is *blocked cuckoo hashing*, where each bucket has two or more slots. This increases the probability of insertion, since we can place an element in any of these slots. It has been theoretically proven that a 2-ary blocked cuckoo hash table with two slots per bucket has a near 100% insertion success rate with a load factor of up to 90% [25]. In this variation, insertions happen serially in each bucket and during evictions, a random slot is evicted.

While both variations limit the number of collisions per insertion and increase the amount of effective space, they also increase the number of accesses for insertions and lookups. Considering this trade-off is a key decision for building a memory efficient cuckoo hashing solution.

2.4 Huge Pages

Huge pages are a memory management feature that allocates memory in larger chunks (usually 2 MiB), instead of the default 4 KiB size. Using huge pages reduces the number of virtual-to-physical translations, improving performance in memory-intensive applications. This alleviates the pressure on the Translation Lookaside Buffer (TLB), a virtual-to-physical address translation cache in the CPU. Since TLBs have a small capacity, with a couple of thousand TLB entries on modern CPUs, huge pages significantly accelerate address translation and minimize TLB misses.

While huge pages have their benefits, they can lead to issues such as memory bloating, increased memory fragmentation and latency, and TLB thrashing. This occurs mostly with applications that have random access patterns, like KV stores and graph processing applications. In reality, popular KV store companies recommend disabling huge pages [6, 17, 19, 22], as their design and mode of operation leads to random memory accesses over the whole address space.

We test if huge pages benefit KV stores by running a benchmark with 4 million SET and GET operations on *redis*, and measuring the reuse distance for each address with Intel's Precise Event Based Sampling (PEBS) [8]. We used 4 KiB and 2 MiB pages (in separate runs), counting the distinct pages

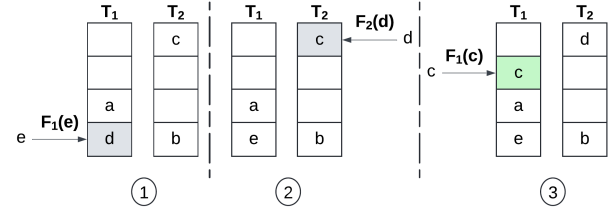


Figure 2. Insertion in cuckoo hash tables.

touched between two accesses to the same page. Our results showed that the majority of 4 KiB pages have good reuse, while the 2 MiB pages vastly surpass the combined size of the TLB. We conclude that 2 MiB pages' benefit for KV stores is limited to page walk effects, rather than coverage.

2.5 Memory Tiering

AutoNUMA and Transparent Page Placement (TPP) are upstreamed memory tiering mechanisms on the Linux kernel. AutoNUMA monitors memory accesses and automatically migrates pages to the NUMA node where they are most frequently used, in an effort to reduce remote memory accesses. TPP focuses more on balancing page placement at allocation time, placing new pages in the "best" NUMA node based on observed access patterns and policies, in an effort to minimize future page migrations.

However, generic memory tiering mechanisms like AutoNUMA and TPP are not suitable for KV stores, since they are designed for homogeneous memory accesses over the whole address space. In contrast, accesses in KV stores are split between two different access types: a compact, contiguous memory space (index) and a random access space (values). Thus, previous tiering mechanisms lead to suboptimal performance, since they treat every access the same.

To show this effect, we use the *redis* and *memcached* indices, with and without resizing, with AutoNUMA and TPP under different memory ratios of DRAM and CXL in our CXL server (Table 3). We use 320 million KV pairs with a 50:50 ratio of SET and GET operations, and show our results in Figure 3.

Figure 3a shows the results without index resizing and Figure 3b contains the results with index resizing. We have the following observations: **1.** Memory tiering negatively affects both *memcached* and *redis*, with *redis* having a more pronounced performance degradation. **2.** SET operations are less affected than GET operations in most cases. We observe that SET operations are affected only in case the whole index cannot fit in a single device after all resizes. **3.** For no resizing, AutoNUMA and TPP lead to worse performance compared to having less DRAM available without memory tiering. **4.** For the resize case, most intermediate results lead to worse performance than placing the entire KV in CXL. This is the

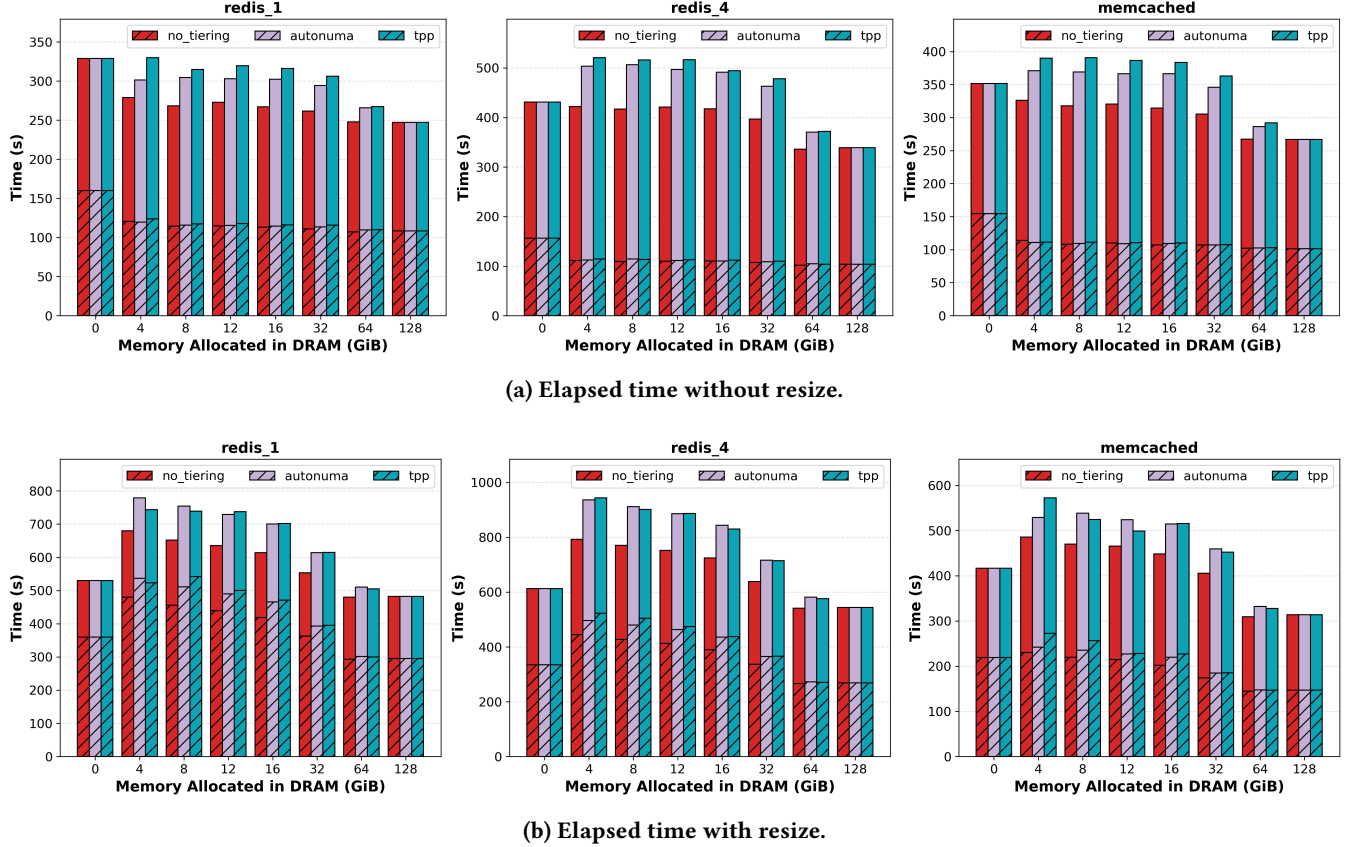


Figure 3. Elapsed time for redis and memcached with different memory tiering configurations. The shaded part represents the SET operations, and the x-axis shows how much DRAM is available, with the rest of the memory allocated in CXL.

Configuration	Time for SETs	Time for GETs	Collisions per Key
2-ary, 2 slots	91.82s	67.51s	0.231
2-ary, 4 slots	85.65s	67.67s	0.074
2-ary, 8 slots	84.56s	70.37s	0.018
3-ary, 1 slot	112.75s	82.11s	0.554
3-ary, 2 slots	101.68s	81.74s	0.22
3-ary, 4 slots	95.41s	80.22s	0.079

Table 1. Cuckoo Hash Table Configurations' Performance.

result of two factors: **(i)** increased cross-device memory accesses and **(ii)** during resizes most of or the whole index is moved from DRAM to CXL, since DRAM is either limited or entirely used by the previous index allocation.

These results clearly show that previous generic memory tiering techniques are inadequate for KV stores. We believe that the right direction is to use KV-specific tiering mechanisms that account for their specific access behavior, and focus on reducing excessive profiling and page migrations.

3 Design

The goal of this work is to build a compact, cache and TLB friendly hash table index that can leverage 2 MiB pages for

numerous key/values. To accomplish that, we have to fit most memory accesses to a small set of 2 MiB pages that will almost guarantee to hit on the TLB for many entries. For this purpose, we use a cuckoo hash table variation, since it provides a compact and performant way to store multiple keys. In this section we describe our index with details about our design decisions.

3.1 Hash Table Entry Structure

To reduce the number of accesses in CBC, we directly store an 8-byte hash of each key, instead of the key itself. This offers two advantages: **(i)** during any operation, we only need a single memory access for the hash, and **(ii)** we reduce the space needed to store large-sized keys. Since the space to store hashes is constant, the reduced memory overhead benefit is proportional to the key size increase. Nevertheless, this approach also creates an issue: during re-insertions we need to rehash the evicted key with the hash function of the next table, in order to find the new position of the entry. To overcome this issue, we store the outputs of all hash functions as the identifier for the key and retrieve the appropriate hash value to figure out the new hash table position.

Since each entry contains the hash values for each hash function and a pointer to the value, the total size of the entry is $(\text{HASH_NUM} + 1) \times 8$ bytes. Typically, a cuckoo hash table has two or three hash functions, so we only need 16 or 24 bytes to store the hashes. In addition, multiple entries can fit into one cache block (usually 64 bytes); for a 2-ary cuckoo hash table we can fit almost three entries (72 bytes) in one cache block, while for a 3-ary cuckoo hash table we can fit two entries (64 bytes) per cache block. On the contrary, existing KV indices need to traverse a linked list to access elements in the same bucket, with each entry access leading to $2\times$ memory accesses.

3.2 Cuckoo Hashing Variations

Using a 2-ary cuckoo hash table limits the usable space, since we are forced to resize when it reaches an occupancy of 50%. Thus, we tested multiple cuckoo hashing variations that guarantee a successful insertion with a load factor close to 90%. The variations that we tested were the following: (i) increasing the number of hash tables, (ii) increasing the slots per bucket (*blocked cuckoo hashing*), and (iii) combinations of both. Table 1 shows the results of a test with 320 million random keys, displaying the time required for insertions and lookups, and the number of collisions per key during insertions.

We notice that increasing the number of hash tables (3-ary) yields the worst performance both for insertions and lookups. Since every key has three possible locations, a 3-ary, 1-slot cuckoo hash table has to pay up to 3 cache line and TLB accesses during lookups. On the other hand, increasing the number of slots per bucket helps speed up both insertions and lookups, since most accesses happen to the same cache lines. In addition, since slots in the same bucket reside in adjacent memory locations, we can access multiple entries in the same cache line and page. For instance, during lookups on a 2-ary hash table, we can search the hashes (16-bytes) of 3 entries by accessing a single cache line.

Regarding blocked cuckoo hashing (multiple slots per bucket) we notice that as the number of slots is increasing, the collision rate drops, while the time for lookups increases with a smaller rate. This is anticipated, since increasing the number of possible locations for each key, decreases the collisions and increases the lookup time. Surprisingly, we notice that the 2-ary, 4-slot cuckoo variation displays similar insertion performance with the 2-ary, 8-slot variation, despite having a higher collision rate. This is the result of having more accesses per bucket for the latter, where more slots per bucket are visited during insertion, leading to a higher cache line miss rate.

In CBC (Figure 5) we use the 2-ary, 4-slot cuckoo hash table as it offers the optimal balance between performant insertions and lookups. Since we use two hash tables, the total entry size is 24 bytes; 2×8 bytes for each hash plus the 8-byte pointer to the value.

3.3 Insertion in CBC

Figure 4 shows how insertion works in CBC. In Step ① we calculate the hash value for both hash functions and try to insert the entry in the 3rd bucket of Table 2. Since it is full, we randomly evict the 2nd element in the bucket (Step ②) and insert the new entry in this slot (Step ③). In Step ④ we retrieve the corresponding bucket in Table 1 for the evicted element ($T_1[F_1(y)]$) and insert the element in the 1st slot, since it is an empty bucket. Finally, in Step ⑤ we show the tables' modified state after the new insertion.

3.4 Keys in CBC & Memory Efficiency

Traditional hash tables with chaining require a dynamic memory allocation scheme. For each collision, a new entry is allocated dynamically and added to the linked list. During deletions, we have to traverse the linked list and then free the entry's memory. Over time, this dynamic allocation and freeing scheme leads to memory fragmentation. On the other hand, all memory in CBC is allocated upfront, with the memory layout remaining contiguous and backed by specific 2 MiB pages. This leads to less pressure on the memory allocator and improves cache and page locality.

As we described in Section 2.1, traditional hash table indices use 24-byte entries just for the pointers to the key, value and next entry. In CBC we save the space for the key by using the 2 hash results instead. Thus, CBC allocates only 24 bytes per entry in total (2×8 -byte hashes + 8-byte pointer to the value), effectively reducing the amount of memory needed. In our tests with 64-byte keys, we notice that CBC uses 37% less memory than redis.

Since CBC utilizes 2 MiB pages to store the index, we dramatically reduce the number of TLB entries needed to access the index. In one of our test server that has 1536 shared TLB entries we can theoretically fit 134 million entries by using 2 MiB pages, while previous approaches that use 4 KiB pages can fit only 262 thousand. Additionally, other systems require one more access for the key, which is not covered by the accesses to the entries themselves as it happens in CBC. Last, our compact approach favors both page and cache locality, further improving our system's performance.

3.5 Hash Collision Probability

Since CBC stores a compact hash identifier in place of the key, a hash collision requires two distinct keys to collide simultaneously under both independent 64-bit hash functions. For a good hash function, the probability that two distinct keys produce the same 64-bit value is $1/2^{64}$; since the two hash functions are independent, the probability that a pair collides under both is $\frac{1}{2^{64}} \times \frac{1}{2^{64}} = \frac{1}{2^{128}}$. This assumes the hash functions are independent and distribute keys uniformly across the hash space, which holds for both of our hash functions, as they pass the SMHasher test suite [2] that tests for these properties.

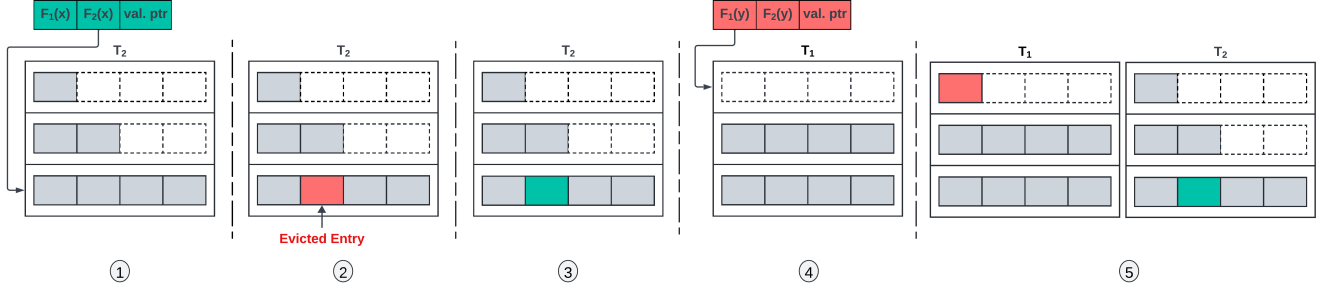


Figure 4. Insertion in CBC.

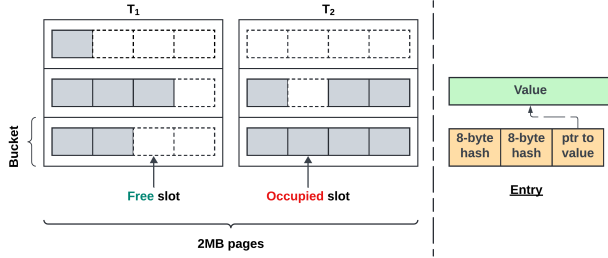


Figure 5. Overview of CBC.

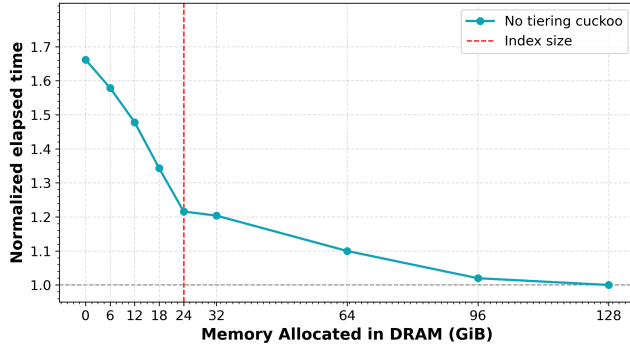


Figure 6. Elapsed time for our cuckoo index in DRAM/CXL.

The number of keys a table can hold before such a collision becomes likely is governed by the birthday bound: with n distinct keys the expected number of colliding pairs is $\binom{n}{2} 2^{-128} \approx n^2/2^{129}$, which reaches one only at $n \approx 2^{64.5} \approx 2.6 \times 10^{19}$ keys. For perspective, a table of one trillion keys has an expected 1.47×10^{-15} false matches, which is less than two chances in a quadrillion. Thus, our 128-bit identifier can serve as a unique key identifier with negligible risk of collision, even for extremely large tables.

3.6 Security

In Section 3.5 we assume that the keys are drawn independently of the hash functions. However, an adversary could choose keys that specifically violate this assumption and

expose CBC to two attacks. The first attack is **hash flooding**, where the adversary chooses keys that collide in both buckets, leading to repeated evictions and insertions, and, in the worst case, a full resize. The second attack is **signature collision**, which is possible only in keyless hash tables like CBC. In this attack, the adversary chooses keys that have the same signature, which leads *GETs* returning wrong values and *SETs* overwriting existing values.

We mitigate these attacks by adding two changes: First, we use secret seeds for both hash functions, which defends against both attacks only under the assumption that the adversary has no access to the key’s signatures. We assume that the latter can be possible, which is why we also feed the output of both hash functions into AES-128 under a third secret key and split the result into two 64-bit identifiers. This hardens CBC and keeps our layout and insertion/lookup path unchanged. The use of secret seeds does not add any overhead, while the use of AES-128 adds a small overhead, as we show in Section 5.4.

3.7 KV Memory Tiering

As we described in Section 2.5, traditional memory tiering approaches for KV stores face significant challenges. To understand the additional memory tiering issues related to our index, we measure the elapsed time with 640 million keys for different DRAM/CXL ratios. We start with all memory in CXL and increasingly add more memory in DRAM until the entire KV fits in DRAM (128 GiB), with memory tiering completely disabled. We show the results in Figure 6 and have the following observations: **(i)** CXL incurs a 65% slowdown compared to the full DRAM run, and **(ii)** the performance does not improve linearly as we add more DRAM: 70% of the total slowdown comes from the index being in CXL. This indicates that by only placing 20% of the total memory size in DRAM we can recover most of the CXL performance slowdown.

To solve the issues of traditional memory tiering approaches on KV stores, we use an approach that allows to efficiently use both tiers of memory. The key idea is to enable memory tiering only for the THP-backed index, which accounts for most accesses and slowdown when deployed in the slower tier. This lowers the profiling and migration overhead, since

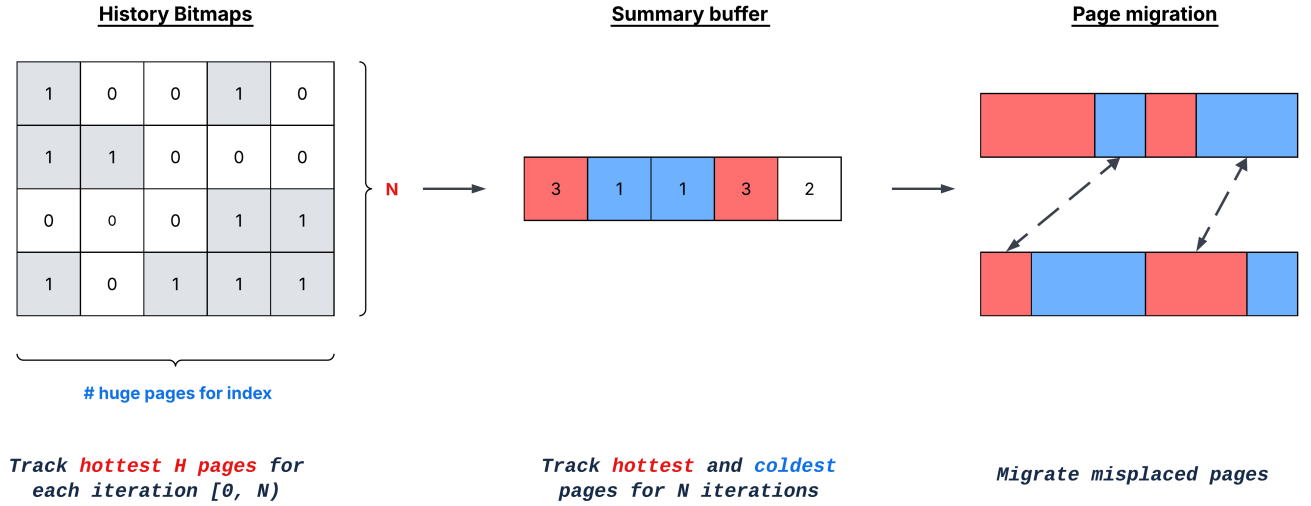


Figure 7. CBC-based memory tiering design.

we monitor and migrate a small fraction of the KV memory space, while optimizing for most slow memory overhead.

We show the design of our memory tiering algorithm in Figure 7. During profiling, we use an N -sized history buffer for our index pages to account for recent and older accesses. This helps to smooth out short-term access spikes and focus on consistently hot pages, which reduces unnecessary migrations. In each profiling iteration we track the top H index pages, where H equals to the number of index pages that fit in DRAM and at the end of the iteration we increase the counter for each corresponding hot page. On each migration iteration we check the summary buffer and migrate the pages that are among the hottest from CXL to DRAM and the coldest that reside in DRAM to CXL.

Since the number of pages to migrate from each device are the same we use the preexisting pages in DRAM and CXL and just swap the hottest with the coldest in each device. This has multiple benefits: **First**, page migrations avoid allocating new pages, which we find comprises roughly 20% of the page migration cost. **Second**, CBC performs page promotions (CXL to DRAM) and demotions (DRAM to CXL) in a single system call, reducing the additional system call overhead. While previous works aim to decouple these two operations, we argue that since DRAM has a fixed size, we always need to demote a page in DRAM to promote a page from CXL.

We introduce three more optimizations that reduce the page migration cost. The first is using multiple threads during page migration; each time we swap two huge pages, we swap concurrently the contents of each huge page in a 4 KiB granularity. The second limits the number of page migrations in a single migration cycle, which limits the migration performance impact to the normal KV operation. Last, we associate each index huge page with a migration epoch threshold MET . This threshold limits re-migrating the

Configuration	Average Collisions per Key	Max Collisions	Execution Time
Always First	0.27	9.8	93.33s
Random	0.075	13.8	89.51s
Interleaving	0.075	12.9	87.88s
Last Inserted	0.074	13.2	85.77s

Table 2. Insertion Policy Performance Metrics.

same page again for the next MET migration cycles, reducing page migration thrashing.

Last, in Figure 3, we showed that during GET operations, memory tiering overhead surpasses the benefits. In CBC we track the ratio of GET and SET operations and disable migrations until the ratio drops below a specific threshold, avoiding the memory migration impact during GET operations.

4 Implementation

This section contains the CBC implementation details, while also providing insights regarding explored approaches.

4.1 Hash Functions

Choosing the right hash function is critical for performance; it should provide a high throughput, good distribution and low collision rates. For this purpose we use *xxHash* and *CLHash*, the two fastest 64-bit hash functions [4, 10] that have been widely tested and used by multiple modern applications, including compression tools, databases, networking tools and Big Data processing frameworks. Both functions successfully complete the SMHasher test suite [2], which evaluates hash functions for collisions and randomness. In addition, both *xxHash* and *CLHash* use Single Instruction, Multiple Data (SIMD) instructions to provide a high throughput for frequent hash operations. Last, for the security part we use hardware-accelerated AES, specifically AES-NI [31],

which is available in both Intel and AMD processors. AES-NI provides fast encryption and decryption, enabling high-performance cryptographic operations.

4.2 Insertion Policy

Minimizing collisions is fundamental for a performant cuckoo hashing scheme. One way to avoid unnecessary collisions is to keep both hash tables equally occupied, to increase the insertion success probability. We test and compare different schemes for the insertion.

We test 4 different schemes: *Always First*, *Random*, *Interleaving* and *Last Inserted*. The first inserts an element always on the first hash table, the second randomly chooses the table, the third interleaves the table to insert and the last one keeps track of the table that got the last inserted element, either by resolving the last collision or by directly accepting the element. For each one of these schemes we measure the average collisions per key, the max number of collisions during insertion, and the total execution time for inserting all elements. We run this over 10 iterations for 320 million keys and report the average results in Table 2.

As expected, inserting always in the first table has the worst performance, since it leads to an unbalanced loading between the tables. On the other hand, the *random* and *interleaving* schemes lead to a more balanced load between the tables, reducing collisions and leading to better performance. Finally, the *Last Inserted* scheme shows the best performance, as it not only balances the load but also directs new elements to the last table where an insertion succeeded. In CBC we use the *Last Inserted* scheme, since it offers the best performance.

4.3 Multithreaded Support

CBC supports concurrent operations through a lightweight, per-bucket synchronization scheme. Since lookups dominate most KV workloads, we optimize for contention-free reads: each bucket has a sequence counter (a *seqlock*) co-located with the bucket’s hash identifiers, so a reader fetches the counter and the identifiers in a single cache-line access. A writer increments the counter to an odd value before modifying the bucket and back to an even value afterwards, while a reader records the counter before its scan and re-checks it after, retrying only if the value changed or is odd. Readers therefore never block and acquire no locks, while writes serialize only at the granularity of the individual bucket they modify. As a result, the per-operation synchronization cost is proportional to the work actually performed, yielding high scalability especially for read-heavy workloads.

4.4 Resizing

In CBC we use a similar resizing policy to that of *redis*: when we surpass a specific load factor (87% in our prototype) we use incremental resizing. We allocate a new cuckoo hash table with double the size and re-insert entries from the old hash table to the new. This operation locks the hash table,

thus we have two conflicting targets: we have to keep it short, to reduce the time it blocks any KV operation, while we have to complete the re-insertion as soon as possible to reclaim the memory from the old cuckoo hash table. To achieve that we limit the partial resize to 10 μ s, which limits the downtime while also achieves a resizing throughput of approximately 12.5 entries per μ s. Resizing our index does not require the keys, since we store both identifiers in each entry and doubling the index size changes only the bucket index, not the signature itself.

4.5 KV memory tiering

For profiling we use DAMON [18], a lightweight, scalable Linux kernel subsystem for application memory profiling. We chose DAMON as it is the only tool that allows monitoring specific memory regions of a process, in our case the index pages. We experimentally selected *N* to be equal to 40 for our history buffers, as it yielded the best trade-off between memory consumption and performance. History buffers are represented as bitmaps that hold one bit per index page, and the summary buffer uses 4-byte counters for each index page. Our goal with these structures is to reduce the profiling memory overhead and TLB and cache use. For instance, for 1 billion keys (index size: 36 GiB), we need roughly 90 KiB for the bitmaps and 72 KiB for the summary buffer.

We implement our profiling and migration logic in the Linux kernel, and use two different threads for profiling and migration. The former updates the DAMON results every 100 ms, while the latter migrates pages every 500 ms. We experimentally confirmed that these numbers offer the best results in CBC, reducing excessive profiling and page migrations. We measured our profiling overhead for different index sizes up to 128 GiB, with the overhead being between 1.8% and up to 2.8%. Last, CBC uses a *Migration Epoch Threshold* of 10, which prevents recently migrated pages to be re-migrated for the next 10 migration epochs.

4.6 Memory Allocation

Since the index is the most performance-critical part of the KV store, CBC prioritizes its allocation to the fast tier, while values are allocated based on the memory allocator policy. If CBC needs to resize the index and there is not enough space in the fast tier, it migrates out values until there is enough space. Apart from that, values are never migrated during normal operation. After testing different allocation and migration schemes we found that migrating values hurts, rather than helps performance.

5 Evaluation

In this section we evaluate CBC and provide answers to the following questions:

Component	Specification
Processor	2 × Intel® Xeon® Silver 4416+
Cores	40 cores (80 threads)
L1d Cache	48 KiB per core
L2 Cache	2 MiB per core
L3 Cache	38 MiB per socket
L1 DTLB (4 KiB)	96 entries
L1 DTLB (2 MiB)	32 entries
Unified STLB	2048 entries
Main Memory	128 GiB DDR5 DIMMs
CXL Support	CXL 1.1

Table 3. CXL Server Configuration.

- What are the hardware performance implications of CBC?
- How does CBC perform compared to widely-used KV stores?
- What is the performance benefit of different page sizes for KV indices?
- How does CBC perform compared to previous memory tiering solutions?

5.1 Workloads and Methodology

For our evaluation we use YCSB-C [24], a C++ version of the YCSB [5] benchmark suite, using five workloads (A-D and F) with different read/write ratio and distributions of keys. Workload E consists of short range scans, which hash-based indices do not support without a secondary ordered structure². We port CBC and compare it against *redis*, which is already included in the benchmark suite.

We also use a set of microbenchmarks that perform SET/GET operations for a variable number of keys and a 128-byte value size, to measure the performance implications of different KVs on the memory subsystem. We run those microbenchmarks for CBC against a minimized version of *redis* and *memcached*, with only their index containing the key and value pointers, with all metadata stripped. Our goal for this is to have a more accurate comparison between the KVs, with a focus on the memory performance implications of their hash table design choices. For *redis* we use two variants: *redis_1* uses resizing when the ratio of entries per bucket is 1, while *redis_4* resizes when the ratio of entries per bucket is 4. These variants are used on the current *redis* deployment and switch automatically to each other depending on the performance impact on resizing.

For the last part of our evaluation, we compare CBC with AutoNUMA, TPP and HybridTier [27], a recent work for memory tiering. HybridTier tracks both short-term momentum and long-term frequency for page migrations and aims to reduce profiling metadata lookups. However, because HybridTier

does not support CXL, we run it on a different server with persistent memory as the slow tier.

Before each run we drop the page cache and compact the memory to ensure accurate measurements between different invocations. We run 5 iterations for our experiments and show the minimum, average and maximum value for each experiment. For each experiment we also use the maximum number of keys that can fit in DRAM without any loss of performance, and disable swap.

5.2 Evaluation Platform

We evaluate CBC on a server equipped with CXL on Linux kernel 6.11.5. Table 3 summarizes the configuration of our server, with information about the CPU, TLB and memory.

5.3 Local Memory KV Performance

This section contains microarchitectural and throughput results where we compare CBC, *redis* and *memcached*. We run all the experiments in local DRAM, without any memory tiering or CXL involvement.

5.3.1 Performance Counters. We use hardware counters to test the KV impact on the memory, CPU caches and TLBs with the use of *perf* [30], a Linux profiling tool that provides a rich set of performance counters. We run the same workload for a variable number of keys, recording each counter in a separate run and reporting the SET (50%) and GET (50%) phases separately.

Cache. During lookups (Figure 8a) CBC produces 4.5 LLC misses per operation, while *redis* reaches up to 10 and *memcached* up to 8.75 LLC misses per operation. This also directly translates to the LLC miss stall cycles, with CBC having up to 2.9× lower stall cycles per operation compared to *redis* and *memcached*. Furthermore, CBC’s misses and stalls are flat across all number of keys, while *redis* and *memcached* display increased numbers as the number of keys increases. This is a direct consequence of chained index dereferences that increase progressively as the size of the index increases.

For insertions (Figure 8b) we notice that CBC displays a higher number of LLC misses per operation, compared to *redis* and *memcached*. However, when we look at the actual performance impact of these misses, i.e. the stall cycles, we notice that CBC has similar performance. While we expected the LLC miss ratio to be higher for CBC, since its index is larger (between 7% and 47% larger), the actual cycles spent for LLC miss resolution shows that CBC is more efficient.

Address Translation. Figure 8c and Figure 8d show the dTLB misses and PTW cycles per SET/GET operation. CBC displays up to 8× lower dTLB LOAD misses per operation and up to 1.8× lower dTLB STORE misses per operation. However, we notice that the PTW LOAD stall cycles are up to 11× lower for CBC, showing that each dTLB LOAD miss has a smaller performance impact on CBC compared to the other KVs. Regarding the PTW STORE stall cycles, we see

²*redis* uses a secondary non-hash index for ranged queries

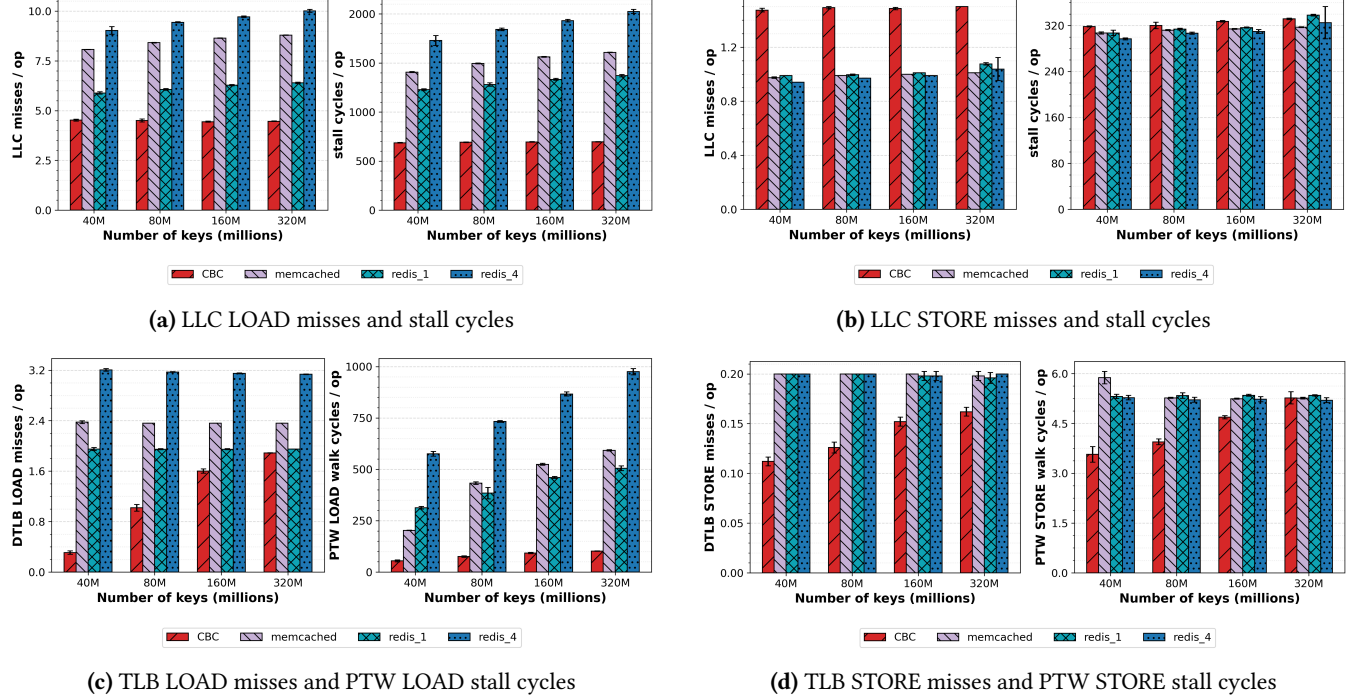


Figure 8. LLC and TLB related performance events.

an equivalent correlation between dTLB misses and PTW stall cycles. In conclusion, we see that the use of huge pages and the density of our index leads to better TLB-related results, with a significant performance benefit, especially for reads/LOAD operations.

Last, we notice that as the number of keys increases, CBC’s TLB counters are increasing per operation, while they remain stable for the other KV. We assume that this behavior is a result of the limited TLB reach in our server’s CPU. We repeat this experiment on another server with an AMD EPYC processor that has more TLB entries. The results showed that the number of TLB misses and PTW stall cycles per operation increase at a much smaller rate for up to 1 billion keys, confirming our assumption.

5.3.2 Multithreaded YCSB-C Performance. Figure 9 shows the throughput of CBC and redis in thousands of operations per second (KTPS) for the five YCSB workloads with up to 80 threads. Both axes are logarithmic; note that the y-axis spans more than two orders of magnitude, so the two systems are separated by a large but slowly varying gap. We use only redis for two reasons: 1. YCSB-C does not provide a memcached binding, and 2. redis and memcached use the same chaining-based index, exhibiting the same memory access behavior. We use the default 23 B keys and 1000 B values with 1 million records and 500 thousand operations, while we repeat each experiment 3 times and record the mean and standard deviation.

Below we present our findings.

CBC sustains a one-to-two orders of magnitude throughput advantage across all workloads. At every thread count and on every workload, CBC outperforms redis by a wide margin. At a single thread CBC delivers between roughly 1.5×10^3 and 4×10^3 KTPS, while redis delivers between 10 and 18 KTPS, a gap of roughly two orders of magnitude. The gap is larger at low thread counts and narrows modestly as threads increase; even at the highest thread counts CBC remains between 30× and 90× faster than redis.

CBC scales with thread count, whereas redis saturates early and regresses. CBC’s throughput rises steadily with additional threads on every workload, continuing to climb through the highest thread counts on the read-heavy workloads (B and C) and reaching its peak of almost 10 thousand KTPS. redis, by contrast, increases only up to approximately four threads, after which its throughput plateaus and then gradually declines toward the 80-thread point. This behavior reflects redis’s single-threaded request processing, while CBC’s per-bucket synchronization allows multiple concurrent operations.

Read-heavy workloads scale best. CBC’s scaling is better on the read-dominated workloads B (95% read) and C (100% read), where throughput grows with thread count and peaks at the 80-thread configuration. On the write-heavier workloads A (50% update) and F (50% read-modify-write), CBC still scales well through moderate thread counts but exhibits a throughput drop at 80 threads, where the cost of bucket-level write serialization and cuckoo eviction begins to outweigh the benefit of additional parallelism. Last, we

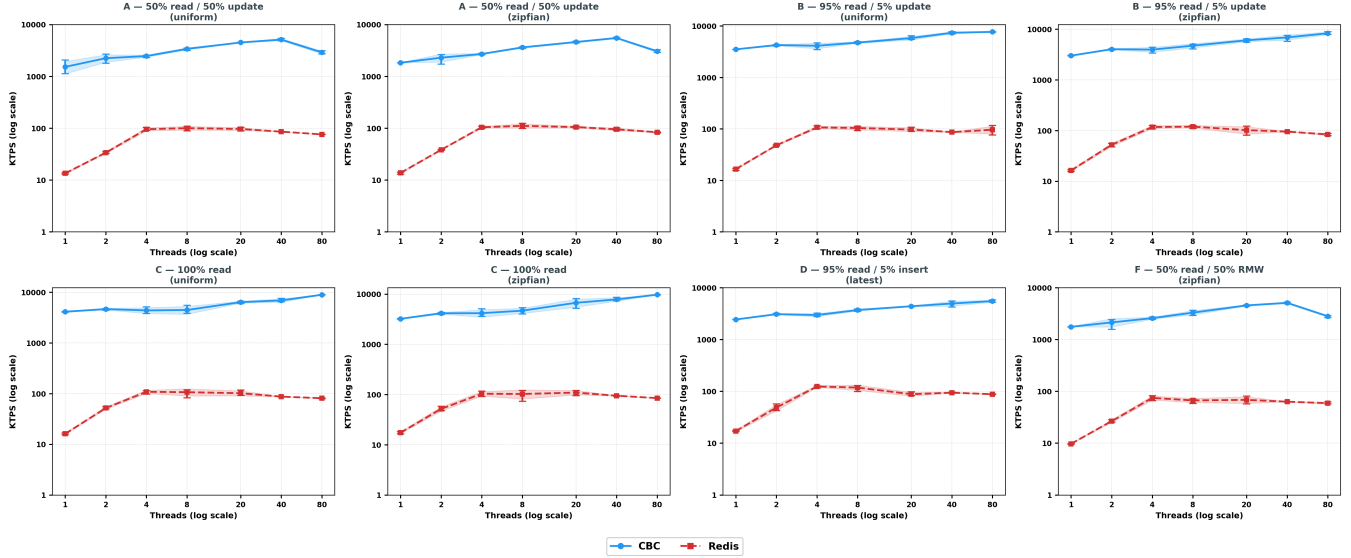


Figure 9. Throughput of CBC vs. redis across YCSB workloads A, B, C, D, F (KTPS; both axes log scale).

Configuration	SET	GET	Total	Coll./key
No security	—	—	—	0.0379
Secret seeds	-0.3%	+0.0%	-0.1%	0.0372
Seeds + AES	+6.3%	+1.4%	+3.5%	0.0836

Table 4. Cost of Hardening. Overhead at 320M keys relative to the unhardened configuration.

notice that the distribution has a comparatively minor effect on both systems.

5.3.3 Page Size Impact & Memory Consumption. We also measure the performance impact that page size has for the index on CBC, redis and memcached, by running our SET/GET microbenchmark with both 4 KiB and 2 MiB pages. While we omit the results due to space constraints, we have three observations: **First**, for all KVs, the use of 2 MiB pages leads to better performance compared to 4 KiB pages, but for redis and memcached this improvement is between 5% and 10%. On the other hand, CBC shows a performance improvement between 13% and up to 22%, compared to the 4 KiB page configuration. **Second**, CBC with 4 KiB index pages still outperforms the rest of the KVs with both index page sizes, showing that our design is more efficient even without the use of huge pages. **Third**, as we increase the number of keys, the performance benefit of using 2 MiB pages becomes more pronounced, especially for CBC. Last, we also measure the memory consumption of every KV, with the results showing that CBC has between 40% and up to 62% lower total memory footprint.

5.4 Security Cost

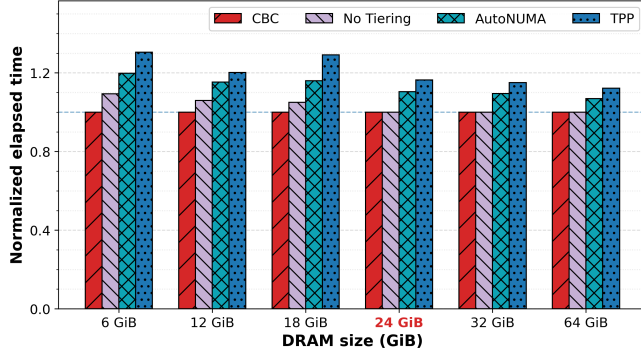
In Table 4 we show the results from the three different configurations with 320 million keys. As expected, using secret

seeds adds no overhead, since it only changes the seed values for the hash functions. Adding AES on top of that introduces a 3.5% overhead in total, with SET operations affected more than GET operations. This is expected, since SET operations can lead to multiple evictions and insertions, requiring more AES operations. We conclude that the security hardening of CBC has a small performance overhead, while preventing both hash flooding and signature collision attacks. Last, we note that the results are very similar for all key counts that we tested.

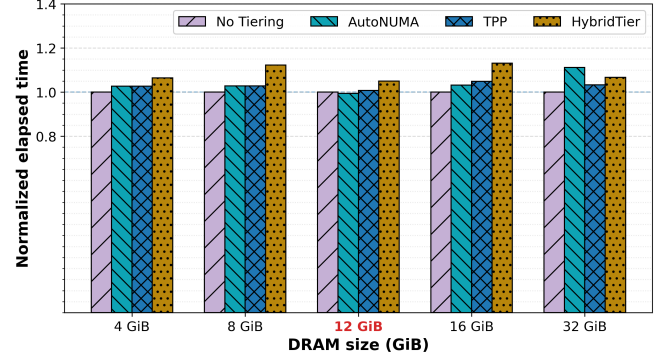
5.5 Memory Tiering Evaluation

We compare CBC with AutoNUMA, TPP and HybridTier and no tiering on two different servers, one equipped with CXL and the other with persistent memory. In HybridTier’s repository [26], the authors state that it does not work out of the box with CXL memory, because its profiler hardcodes PEBS events that cannot capture CXL accesses. On all CXL machines that we tried with HybridTier’s configuration and suggested changes, the CXL-attached memory could not be identified as an additional NUMA node. Thus, we compare CBC with all configurations but HybridTier on our CXL server, and then compare HybridTier on our persistent memory server with all configurations but CBC, since we developed CBC on a newer kernel version than those supported by this server. Last, we disable CBC’s memory tiering when we run the other tiering mechanisms.

5.5.1 Memory Tiering on CXL. We use 640 million keys and do an equal number of SET and GET operations over the entire key space, and increasingly use more DRAM until a little over 50% of the KV’s memory fits in DRAM. The index size for 640 million keys is 24 GiB, so CBC automatically disables memory tiering when the entire index is in DRAM.



(a) Memory tiering results on CXL for CBC, no tiering, AutoNUMA and TPP. The index size is 24 GiB.



(b) Memory tiering results on PMEM for no tiering, AutoNUMA, TPP and HybridTier. The index size is 12 GiB.

Figure 10. Memory tiering results. The y-axis shows the normalized execution time, with lower being better.

Figure 10a includes the CXL memory tiering results, with CBC outperforming both the other tiering mechanisms and no tiering. Specifically, CBC has between 6% and up to 10% better performance over no tiering and 18% and performs up to 30% better compared to the other tiering mechanisms.

Our performance improvement is a result of multiple design decisions: We only profile and consider for page migration the index pages, which reduces both profiling and migration overhead. Additionally, migrating a single index page on CBC covers a larger portion of the index, which increases the probability of reusing the migrated pages. We further reduce the migration overhead by limiting the number of migrations and using the migration epoch threshold.

On the contrary, AutoNUMA and TPP display worse performance than no tiering. Since both mechanisms do not distinguish between the index and key/values parts of the address space, they display increased profiling and memory migration costs, which surpass the performance benefit. As expected, TPP exhibits worse performance, since it uses more aggressive profiling and migration policies than AutoNUMA.

Last, even with a larger DRAM-to-CXL ratio (not shown), no tiering outperforms AutoNUMA and TPP, with the performance overhead decreasing as the DRAM increases. For roughly 90% of the KV in DRAM, AutoNUMA and TPP show 3% and 5% performance overhead, respectively.

5.5.2 Memory Tiering on Persistent Memory. We repeat the same experiment with 320 million keys on our Persistent Memory server (Table 5), with a 12 GiB index. We notice that AutoNUMA and TPP exhibit better relative performance to no tiering, compared to our CXL experiments. This was expected, since we use fewer keys, which leads to a smaller overhead. Overall, AutoNUMA has between 1.01× and up to 1.11× overhead over no tiering, with TPP exhibiting between 1.01× and up to 1.05× performance overhead.

HybridTier displays the worst performance among the tiering mechanisms, with a performance overhead between

Component	Specification
Processor	Intel® Xeon® Gold 6230
Cores	20 cores (40 threads)
L1d Cache	32 KiB per core
L2 Cache	1 MiB per core
L3 Cache	27.5 MiB per socket
L1 DTLB (4 KiB)	64 entries
L1 DTLB (2 MiB)	32 entries
Unified STLB	1536 entries
Main Memory	96 GiB DDR4 DIMMs
Persistent Memory	Intel Optane

Table 5. Persistent Memory Server Configuration.

1.05× and up to 1.14×. While HybridTier uses both short-term burst and long-term frequency to profile and migrate pages, it still suffers from excessive profiling overhead and over-migrating pages that are not frequently used. We confirm the former by enabling HybridTier when all memory is placed in DRAM, with zero migrations happening during execution; HybridTier’s profiling mechanism alone introduced between 1.16× and up to 1.20× performance overhead.

While we do not directly compare CBC with HybridTier (since it does not support CXL), we can safely assume that CBC outperforms it. HybridTier displays a worse performance compared to AutoNUMA and TPP, while CBC outperforms them by up to 30%.

5.5.3 Memory Tiering Summary. Overall, our results show that previous memory tiering mechanisms fail to offer any performance benefit, while CBC reduces the profiling and migration overhead and offers increased tiering performance. Our improvement over no tiering at all is best read against what tiering normally offers rather than against the full-DRAM ceiling: we show that memory tiering for KV stores is possible and worth enabling. This near-DRAM throughput is achieved with substantially less DRAM; because DRAM

accounts for up to 50% of server cost, recovering most of the performance at a fraction of the DRAM footprint improves cost-efficiency. This is extremely important for production memory management at scale, since at hyperscale even fractional memory savings compound into enormous absolute reductions in DRAM capacity, power, and cost [29].

6 Related Work

This section contains the related work regarding KV stores and memory tiering solutions.

6.1 Key-Value Store Architectures

Key-value stores fall into two main categories: log-structured merge (LSM) tree-based and hashtable-based. LSM-based stores, such as RocksDB [14], are hybrid memory/storage KVs that organize data in sorted levels and rely on sequential writes and background compactions to provide efficient storage management and range-query support. In contrast, hashtable-based systems like Redis [23] and Memcached [13] focus on fast point lookups and updates, achieving sub-millisecond latency by storing keys directly in memory with collision-resolution mechanisms.

Hashtable-based in-memory KV stores have been extensively adopted for latency-critical services, offering fast accesses through hash tables. While these systems achieve high throughput, their index structures impose significant memory overhead due to pointers, metadata, and collision resolution mechanisms. Prior optimizations have focused on reducing latency and distributed scalability across clusters. However, less attention has been paid to index memory efficiency, especially under heterogeneous memory systems. Our work complements these systems by redesigning the index itself: CBC eliminates full key storage, reduces per-entry memory overhead, and utilizes 2 MiB pages, providing a compact, TLB-friendly and cache-friendly design.

6.2 Memory-Efficient KV Indices

Prior work reduces index footprint by using a short identifier before touching the key, but retains the key for verification. MemC3 [7] and libcuckoo [11] use an 8-bit tag per slot to avoid pointer dereferences, while FASTER [3] packs tag bits into its bucket entries. However, all three confirm a tag match against the full key. SILT [12] has an average overhead of 0.7 bytes per key in DRAM (less than CBC), but stores the full key on flash in an immutable sorted store, and verifies it once the record is read. ZStore [28] similarly uses a key hash for its in-memory index, but retains the full key in the on-device object header. Segment-structured and Persistent Memory designs [33, 35, 36] likewise retain keys. In contrast, CBC avoids storing the keys, replacing each key with a 128-bit identifier, enabling its compact, memory-efficient design.

6.3 Memory Tiering and Heterogeneous Memory Systems

The ever-growing existence of memory-hungry workloads has made memory tiering a necessity for limiting cost and increasing performance. Recent studies [1, 29] have shown that the cost of DRAM accounts for up to 50% of the servers' cost, while a significant portion of memory in DRAM is not frequently used. These insights require revising old and developing new mechanisms that reduce DRAM usage, increasing allocated memory utilization and reducing the total server costs. The advent of heterogeneous memory technologies, such as CXL, has introduced new research opportunities with multiple systems [9, 15, 16, 20, 27, 34] leveraging the benefits of different memory technologies.

HybridTier [27] tiers data between two tiers by tracking both long-term access frequency and short-term momentum with compact blocked counting Bloom-filter metadata, promoting hot pages to DRAM and demoting cold ones with low overhead. Memtis [9] combines page placement and page size use, using huge pages for extremely hot pages and improving locality. HeMem [20] uses memory access sampling and migrates pages asynchronously in batches, reducing TLB-shutdown and tracking overheads for big-memory workloads. NOMAD [34] introduces non-exclusive memory tiering that temporarily retains copies across tiers by using transactional page migration and page shadowing. While these systems work relatively well for generic applications, their profiling overhead, migration granularity and KV random access patterns limit their effectiveness for KVs.

7 Conclusion

We present CBC, a keyless, compact and memory-efficient key-value index tailored for heterogeneous memory systems. CBC leverages 2 MiB-backed blocked cuckoo hashing that eliminates the need to store full keys, sustaining high load factors while significantly improving TLB coverage and cache locality. We introduce a tiering-aware memory management mechanism that focuses migration and profiling exclusively on the index, allowing CBC to mitigate the penalties of slower memory tiers, while reserving DRAM for critical data. Our evaluation shows that CBC outperforms modern KV stores and previous memory tiering techniques, while significantly reducing the total memory overhead.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable feedback. This paper is supported in part by the PRISM center within JUMP 2.0, an SRC program sponsored by DARPA.

References

- [1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020.

- Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. <https://www.usenix.org/conference/nsdi20/presentation/agache>
- [2] Austin Appleby. 2026. SMHasher. Retrieved October 2026 from <https://github.com/aappleby/smhasher>
- [3] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, Houston, TX, 275–290. doi:10.1145/3183713.3196898
- [4] Yann Collet. 2026. xxHash - Extremely fast hash algorithm. Retrieved October 2026 from <https://github.com/Cyan4973/xxHash>
- [5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (Indianapolis, Indiana, USA) (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152
- [6] Couchbase. 2026. Disabling Transparent Huge Pages (THP). Retrieved October 2026 from <https://docs.couchbase.com/server/current/install/thp-disable.html>
- [7] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Lombard, IL, 371–384.
- [8] Intel Corporation. 2026. Intel® 64 and IA-32 Architectures Software Developer Manuals. Retrieved October 2026 from <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [9] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (Koblenz, Germany) (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 17–34. doi:10.1145/3600006.3613167
- [10] Daniel Lemire and Owen Kaser. 2015. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering* 6, 3 (Sept. 2015), 171–185. doi:10.1007/s13389-015-0110-5
- [11] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*. ACM, Amsterdam, The Netherlands, Article 27, 14 pages. doi:10.1145/2592798.2592820
- [12] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Cascais, Portugal, 1–13. doi:10.1145/2043556.2043558
- [13] Memcached Open-Source Community. 2026. Memcached. Retrieved October 2026 from <https://memcached.org/>
- [14] Meta Platforms, Inc. 2026. RocksDB. Retrieved October 2026 from <https://rocksdb.org/>
- [15] Theodore Michailidis, Juno Kim, Linsong Guo, Steven Swanson, and Jishen Zhao. 2025. TOSS: Tiering of Serverless Snapshots for Memory-Efficient Serverless Computing. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2–14. doi:10.1109/IPDPS64566.2025.00010
- [16] Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2022. PMSifter: Enabling Persistent Memory Fluidness in Linux. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems (Virtual Event, Singapore) (APSys '22)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3546591.3547523
- [17] MongoDB. 2026. Disable Transparent Huge Pages (THP). Retrieved October 2026 from <https://www.mongodb.com/docs/manual/tutorial/transparent-huge-pages/>
- [18] SeongJae Park. 2026. DAMON: Data Access Monitor. Retrieved October 2026 from <https://sjp38.github.io/post/damon/>
- [19] Percona. 2026. Transparent Huge Pages Refresher. Retrieved October 2026 from <https://www.percona.com/blog/transparent-huge-pages-refresher/>
- [20] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (Virtual Event, Germany) (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. doi:10.1145/3477132.3483550
- [21] Redis. 2025. Keyspace - Redis Documentation. Retrieved October 2026 from <https://redis.io/docs/latest/develop/use/keyspace/>
- [22] Redis. 2026. Diagnosing latency issues. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/latency/ [Online; Accessed October, 2026].
- [23] Redis Ltd. 2026. Redis. <https://redis.io/> [Online; Accessed October, 2026].
- [24] Jinglei Ren. 2026. YCSB-C: A C++ version of YCSB. Retrieved October 2026 from <https://github.com/basicthinker/YCSB-C>
- [25] Tim Roughgarden. 2015. Lecture 11: Cuckoo Hashing. Retrieved October 2026 from <https://web.stanford.edu/class/cs166/lectures/11/Small11.pdf>
- [26] Kevin Song. 2025. Artifact repository for HybridTier (ASPLOS 25). Retrieved October 2026 from <https://github.com/kevins981/hybridtier-asplos25-artifact>
- [27] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. 2025. HybridTier: an Adaptive and Lightweight CXL-Memory Tiering System. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Rotterdam, Netherlands) (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 112–128. doi:10.1145/3676642.3736119
- [28] Shuwen Sun, Isaac Khor, Ji-Yong Shin, and Peter Desnoyers. 2026. A Fast, Efficient, and Strongly-Consistent Object Store. In *Proceedings of the 2025 ACM Symposium on Cloud Computing (Online, USA) (SoCC '25)*. Association for Computing Machinery, New York, NY, USA, 708–721. doi:10.1145/3772052.3772272
- [29] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731
- [30] Wikipedia contributors. 2022. Perf (Linux) — Wikipedia, The Free Encyclopedia. [https://en.wikipedia.org/w/index.php?title=Perf_\(Linux\)&oldid=1126307013](https://en.wikipedia.org/w/index.php?title=Perf_(Linux)&oldid=1126307013) [Online; Accessed October, 2026].
- [31] Wikipedia contributors. 2026. AES instruction set. Retrieved October 2026 from https://en.wikipedia.org/wiki/AES_instruction_set
- [32] Wikipedia contributors. 2026. Cuckoo Hashing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Cuckoo_hashing [Online; Accessed October, 2026].
- [33] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*. USENIX Association, Santa Clara, CA, 349–362.
- [34] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 19–35. <https://www.usenix.org/conference/osdi24/presentation/xiang>

- [35] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. Segcache: A Memory-Efficient and Scalable In-Memory Key-Value Cache for Small Objects. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 503–518.
- [36] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, 461–476.