TaC: An Anti-Caching Key-Value Store on Heterogeneous Memory Architectures

Yunhong Ji
Renmin University of China
China
jiyunhong@ruc.edu.cn

Wentao Huang
National University of Singapore
Singapore
huang@comp.nus.edu.sg

Xuan Zhou
East China Normal University
China
xzhou@dase.ecnu.edu.cn

Bingsheng He
National University of Singapore
Singapore
hebs@comp.nus.edu.sg

Kian-Lee Tan
National University of Singapore
Singapore
tankl@comp.nus.edu.sg

ABSTRACT

In-memory key-value (KV) stores play a pivotal role in modern applications due to their exceptional performance. However, they grapple with the high cost and limited capacity of DRAM. Anti-caching systems address these limitations by using the disk (or SSD) to store cold data evicted from memory. However, as data volumes surge, the performance of anti-caching systems can degrade significantly. Luckily, the emerging byte-addressable storage, such as Non-Volatile Memory (NVM), offers larger capacity and enhanced cost-effectiveness compared to DRAM. This paper delves into its potential in building anti-caching KV stores for large-scale data.

Due to the performance degradation of NVM compared to DRAM and its specific performance characteristics, how to efficiently integrate it into an anti-caching KV store poses challenges. In this paper, we discuss several potential designs and propose a three-tier anti-caching design, TaC. TaC utilizes NVM to expand the memory capacity of anti-caching systems and employs DRAM, NVM, and SSD to host hot, warm, and cold data, respectively. In particular, the three-tier architecture introduces additional challenges in data swapping and access tracking. To address them, we introduce a lightweight access tracking mechanism and a hybrid data swapping strategy. We implemented a prototype of TaC on top of the widely-used open-source in-memory KV store Memcached and evaluated it using the YCSB benchmark. The results demonstrate that TaC can outperform alternative designs across various workloads.

1 INTRODUCTION

State-of-the-art in-memory Key-Value (KV) stores, such as Redis [39] and Memcached [30], play a crucial role in modern applications [52]. One typical use case is the application-level cache [31], which stores encapsulated results of frequently invoked application methods operating over databases, effectively reducing the backend system workload. Therefore, a KV store with both large capacity and excellent performance is of paramount importance. However, the scalability of current in-memory KV stores is often limited by the high cost and capacity restrictions of DRAM [23]. The typical solution to expand capacity is to establish a cluster, which can be expensive and complex. In this paper, we explore an efficient method for expanding the KV store capacity on a single machine, a solution that can also benefit cluster setups by allowing more data to be stored with fewer machines.

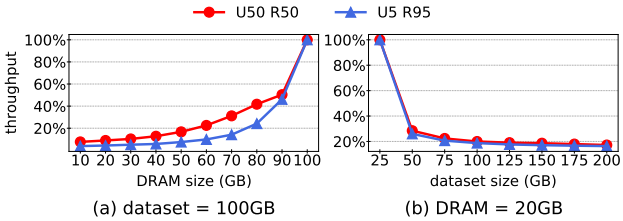


Figure 1: Normalized performance of FASTER with different data volume and DRAM configuration, where "Ux" and "Ry" refer to x% Update and y% Read requests in the YCSB benchmark.

Anti-caching, as proposed in the previous work [11], is an effective mechanism for expanding the capacity of in-memory systems. It incorporates a slower but cost-efficient storage medium, such as a disk or SSD, referred to as an anti-cache, to store cold data from memory. Typically, it can achieve performance close to that of the in-memory system through the following strategies:

(1) Single Copy: Anti-caching systems store data in a single copy, eliminating redundancy across different storage tiers. This reduction in redundancy not only conserves resources but also minimizes data synchronization overhead [11].
(2) Fine-Grained Eviction: To optimize memory hit rates, anti-caching employs fine-grained eviction techniques. It collects cold in-memory tuples and organizes them into blocks for eviction to the disk or SSD. It is more conducive to maintaining high memory hit rates compared to the block-level data swapping commonly used in caching systems [53].
(3) Asynchronous Fetching: Anti-caching employs asynchronous fetching, removing disk (or SSD) I/O operations from the critical path. This helps to mitigate the waiting time for I/O operations, enhancing overall system throughput [11].

While anti-caching proves highly effective when the majority of data resides within the memory, its performance diminishes when hot data surpasses the available memory size. This is because the increase in disk activity can lead to noticeable performance degradation, as noted in a previous study [53]. In Figure 1, we present the performance of FASTER, an anti-caching style Key-Value store, with the YCSB benchmark [9] and a Zipfian request distribution. As illustrated, FASTER experiences substantial performance degradation as the ratio of $\frac{DRAM\ size}{dataset\ size}$ decreases. Specifically, its performance drops to as low as 20%

Table 1: Capacity and prices ($/GB) of different memory.

Capacity	128GB	256GB	512GB
DDR5 DRAM DIMM [12]	11.3	12.5	-
Intel Optane PMem [32]	8.6	8.4	8.2

when $\frac{DRAM\ size}{dataset\ size}$ decreases from 100% to 50%. This performance challenge becomes increasingly critical due to the growing gap between the rapid expansion of data size and the slower growth of DRAM capacity [23].

Fortunately, emerging byte-addressable storage technologies, such as Non-Volatile Memory (NVM) [1, 37, 50], offer a cost-effective solution for expanding memory capacity [38]. These technologies are primarily designed to overcome the scaling limitations of DRAM [17], providing larger capacities and near-DRAM performance at a lower cost. In particular, Table 1 presents the capacity and prices of recent server DRAM and Intel Optane NVM DIMMs. It highlights the advantages of NVM which offers significantly lower cost-per-GB and higher capacities compared to traditional DRAM. Despite the suspension of Intel Optane NVM, the cost-effectiveness of future NVM products is expected to persist due to their high density [17]. In light of these developments, this paper delves into the exploration of harnessing the potential of NVM to optimize anti-caching Key-Value stores.

While the byte-addressable nature of NVM suggests an intuitive solution: utilizing NVM in the same way as DRAM to expand the memory capacity of existing anti-caching Key-Value stores, this approach faces significant challenges due to the performance disparities between NVM and DRAM, as extensively discussed in prior researches [13, 49]. Besides, NVM exhibits distinct performance characteristics, particularly concerning access patterns and limited bandwidth, which become particularly pronounced in high-concurrency scenarios [2]. As a result, the direct substitution of DRAM with NVM often yields unexpected performance variations.

Therefore, efficiently combining NVM and DRAM into a unified heterogeneous memory presents challenges, and identifying an effective solution for incorporating NVM into anti-caching Key-Value stores is of paramount importance. In this paper, we explore several potential approaches and introduce an efficient three-tier anti-caching Key-Value store, which we named TaC. TaC is meticulously designed to accommodate the specific characteristics of NVM by treating DRAM, NVM, and SSD as distinct tiers, each dedicated to storing hot, warm, and cold data, respectively.

A three-tier architecture, however, introduces several challenges for an anti-caching system. Firstly, it brings additional data swapping paths into the equation compared to conventional two-tier architectures [56]. The careful selection of these paths is crucial, taking into account the unique characteristics of each storage device. In response, TaC employs a hybrid data swapping strategy that optimizes these paths, thereby enhancing SSD I/O utilization.

Secondly, the three-tier system, compared to traditional two-tier ones, requires a multi-level data classification due to more choices in placing the data. Further, the finer-grained data access tracking it necessitates can incur more overhead. To address this challenge, TaC introduces "Lazy LRU," a lightweight tuple-level access tracking mechanism. It defines multiple temperature levels to guide data swapping processes and allows batch processing of updates on the LRU list, a measure that significantly reduces overhead.

In summary, we made the following contributions:

1) To the best of our knowledge, this is the first work that explores the designs of anti-caching systems on heterogeneous memory. We discuss several potential designs for integrating NVM into anti-caching KV stores and propose an efficient three-tier design.

2) We introduce an efficient design for anti-caching KV stores that utilize DRAM, NVM, and SSD, simultaneously. This design leverages a hybrid data swapping strategy and "Lazy LRU" to address the challenges associated with the three-tier architecture.

3) We have implemented our prototype and alternative designs on top of a widely-used open-source in-memory Key-Value store, Memcached [30]. We conducted extensive experiments using the YCSB benchmark [9]. The results confirm that, in comparison to alternative designs, the three-tier architecture is the most efficient approach to integrating NVM into anti-caching KV stores.

The rest of this paper is organized as follows: In Section 2, we introduce the background and related works. Section 3 discusses potential anti-caching architectures when integrating NVM and Section 4 provides a high-level design and the technical choices made by TaC. Then, implementation details of the system are presented in Section 5. The results of the experimental study are reported in Section 6. Finally, there is a conclusion of the paper in Section 7.

2 BACKGROUND AND RELATED WORK

In this section, we first introduce the studies about anti-caching. Then, we summarize the related works of KV stores on NVM and hybrid storage utilizing NVM. Finally, we provide a discussion about future byte-addressable devices.

2.1 Anti-caching Systems

Anti-caching systems have been introduced as a solution to expand the capacity of in-memory systems [11]. Their primary objective is to enhance performance by offloading cold data to an anti-cache while retaining hot data in DRAM, thereby enabling the system to handle most requests as efficiently as an in-memory system. Since the inception of anti-caching, extensive research has delved into its mechanisms and components, aiming to uncover the key factors that contribute to its performance. Notably, a comprehensive survey [53] has systematically summarized and compared various implementation strategies, emphasizing the effectiveness of tuple-level data swapping in optimizing memory utilization.

In recent years, there have been some studies [8, 24, 29] to discuss KV stores on hybrid DRAM and SSD architecture. Specifically, FASTER [5] is a Key-Value store developed by Microsoft Research that exhibits anti-caching-like characteristics. It operates by managing both disk and DRAM as a unified logical address space and offloading older data to disk when the DRAM reaches its capacity threshold. FASTER employs several performance optimization techniques, including a latch-free concurrent hash index and an in-place update mechanism. When the dataset fits within DRAM, FASTER can achieve performance levels comparable to or better than traditional in-memory systems.

However, due to the distinct characteristics of NVM and DRAM, optimization strategies that work well for DRAM-based anti-caching systems may not yield the same results when applied to NVM. Our experimental evaluation, detailed in Section 6.2,

demonstrates that a two-tier anti-caching system that simply treats NVM as if it were DRAM fails to deliver acceptable performance.

2.2 KV Stores on NVM

In recent years, a significant number of works have been devoted to optimizing KV stores on NVM. Some attempted to rebuild conventional data structures on NVM, such as B-trees or B+ trees [14, 25], hash tables [16, 28, 34], and LSM-trees [51, 54]. A significant body of works [3, 10, 15, 19, 44, 46] considered how to integrate DRAM and NVM into an efficient heterogeneous memory. Some works use DRAM to hold the most frequently accessed data structures, such as indexes, to hide the latency of NVM [3, 6, 15, 33]. Some works use DRAM as a cache or buffer of NVM to speed up data access [36, 44]. However, they typically assumed that everything could reside in NVM without considering the use of SSD as an anti-cache.

Different from these works, we argue that it may not be cost-effective to maintain the entire data set in NVM, considering the rapid growth of data volume in modern applications [23]. In the foreseeable future, SSDs and hard disks are expected to remain the primary storage solutions for mainstream applications. As a result, we consider a three-tier storage architecture, which employs SSD as an anti-cache, to achieve a more realistic tradeoff between capacity and performance.

2.3 Hybrid Storage Utilizing NVM

Besides KV stores, using NVM to expand the capacity of DRAM has been widely studied. HeMem [38] and HSCC [26] manage NVM and DRAM as unified memory space, but without considering SSD. HYMEM [43] and Spitfire [56] treat both DRAM and NVM as caches (or buffers) of SSD. Prism [42] utilizes DRAM as the read cache and NVM as the write buffer. Kassa et al. [22] employ NVM to enlarge the memory tier of RocksDB [41]. They are all dedicated to architectures that treat HDD or SSD as the primary storage while anti-caching is designed to extend in-memory systems. Besides, Ziggurat [55] is a file system that allows data to span and be swapped between NVM and SSD, but without considering DRAM.

Since the performance disparity between NVM and DRAM is relatively small [38], the overhead associated with data swapping becomes more pronounced. Specifically, OAM [27] implements object-level memory management utilizing a profiling tool that examines the source code of applications. However, this approach may not be suitable for a general KV store, as the source code of applications is often unavailable.

In summary, none of the above studies has discussed the anti-caching architecture based on NVM. To the best of our knowledge, TaC is the first anti-caching system that leverages NVM to extend the memory and retains SSD as a supplementary.

2.4 Future Byte-addressable Storage

The decision by Intel to wind down its Optane DIMM business [20] has sparked concerns regarding the future of NVM research. However, we firmly contend that NVM technology, initially conceived to address the scaling limitations of DRAM [17], continues to be relevant and essential.

For clarity, we make the following performance assumptions about NVM that serve as the guiding principles for our design:

- *Inferior performance* ($\mathcal{P}_1$). NVM exhibits a noticeable performance gap when compared to DRAM [13], potentially leading to unexpected performance variations.
- *Read-write asymmetry* ($\mathcal{P}_2$). NVM displays read-write asymmetry concerning latency and bandwidth, potentially making write operations a performance bottleneck [13].
- *Inferior performance on small and random accesses* ($\mathcal{P}_3$). NVM suffers from a coarser access granularity, resulting in the well-known read/write amplification issue for small and random data requests [13]. This emphasizes the importance of data locality.
- *Limited concurrency* ($\mathcal{P}_4$). NVM exhibits specific concurrency constraints [2], particularly for write operations. An excessive number of writing threads can lead to performance degradation.
- *Interference with DRAM* ($\mathcal{P}_5$). High concurrency access to NVM can negatively impact the bandwidth of DRAM [45]. The extent of degradation increases as the number of threads accessing NVM simultaneously rises.

These assumptions are primarily based on existing NVM devices like Optane [47] and the widely adopted NVM standard, NVDIMM-P [21], which organizes NVM as memory DIMMs attached to the memory bus. We expect that future NVM products will adhere to this standard and share similar characteristics [17].

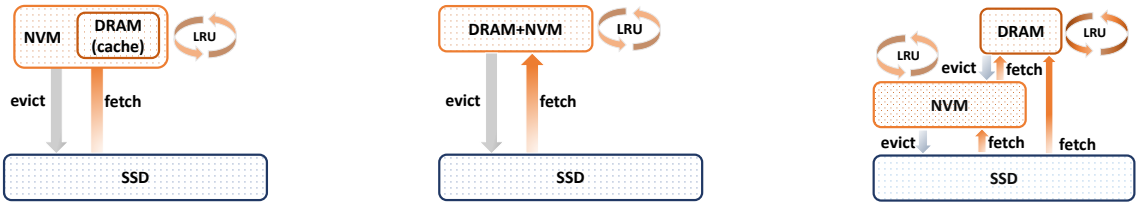
Moreover, our architectural approach can be adapted to other byte-addressable devices, which are anticipated to share similar performance characteristics [2]. The emerging CXL standard [7], for instance, which trades off access latency to expand memory capacity, introduces new possibilities for heterogeneous memory architecture. The existing performance gap between CXL memory and DRAM, akin to that between NVM and DRAM [4, 18], indicates that the discussion in this paper remains valuable.

3 ANTI-CACHING WITH NVM

In this section, we discuss several potential designs to integrate the NVM into an anti-caching architecture. In particular, when considering an anti-caching system with DRAM, NVM, and SSD, there could be three primary architecture options as illustrated in Figure 2:

- the two-tier architecture in Figure 2(a), replacing DRAM with NVM while utilizing DRAM as the cache of NVM;
- the two-tier architecture in Figure 2(b), which utilizes NVM in the same way as DRAM;
- the three-tier architecture in Figure 2(c), utilizing NVM as the middle tier between DRAM and SSD.

Firstly, in the realm of utilizing NVM to expand memory capacity, two traditional architectures have emerged [13]: "Memory Mode" and "AppDirect Mode". The "Memory Mode" architecture replaces DRAM with NVM while treating DRAM as a cache for NVM to handle hot data efficiently. In contrast, the "AppDirect Mode" architecture treats NVM and DRAM as distinct memory spaces, allowing programmers to manage them separately. Therefore, when considering the integration of NVM into an anti-caching Key-Value store, the initial choice is to embrace the "Memory Mode" architecture. In this configuration, DRAM is replaced with NVM, and NVM serves as the primary storage, while SSD is employed as the anti-cache. DRAM, in this context, functions as a cache for holding hot data in NVM. This architecture can be easy to implement such as directly applying the "Memory Mode" of Optane. However, in this design, data in DRAM exists in duplicate, necessitating its transfer to NVM before eviction.



(a) Two-tier design with DRAM as a cache. (b) Two-tier design not distinguishing DRAM and NVM. (c) Three-tier design (TaC).

Figure 2: Potential architectures of anti-caching-based KV-stores with DRAM, NVM, and SSD.

The presence of two copies of data in DRAM and NVM can result in suboptimal memory utilization.

The second design represents an extension of the classical anti-caching system, where NVM and DRAM are not treated distinctly. In this architecture, DRAM and NVM are merged into the primary storage, while SSD serves as the secondary storage. This design is also typically easy to implement and can be derived directly from existing two-tier anti-caching systems. Further, it could also solve the memory waste problem of the first design. However, it does not differentiate between the characteristics of NVM and DRAM. Due to the read-write asymmetry ($\mathcal{P}_2$) and the higher overhead associated with small and random access ($\mathcal{P}_3$) in NVM, using NVM in the same manner as DRAM can result in inefficient utilization of NVM. The performance gap between NVM and DRAM ($\mathcal{P}_1$) further exacerbates this issue due to the suboptimal DRAM utilization, leading to diminished system performance.

The final design, as implemented in our three-tier anti-caching architecture of TaC, distinguishes between DRAM, NVM, and SSD. In this arrangement, DRAM serves as the top tier for hot data, SSD as the bottom tier for infrequently used data, and NVM as the middle tier for warm data. This design offers several advantages:

(1) *Leveraging DRAM Strengths.* By distinguishing between DRAM and NVM and allocating hotter data to DRAM, we can harness the strengths of DRAM more effectively. This approach enhances the DRAM hit rate, reducing the need to access NVM. Consequently, it mitigates the impact of high latency ($\mathcal{P}_1$) and limited bandwidth ($\mathcal{P}_4$) of NVM on system performance.

(2) *Hot Data Retention.* NVM provides hot data with another opportunity to return to DRAM before being transferred to SSD. This mechanism improves the differentiation between hot and cold data, reducing the influence of misjudging data temperature.

(3) *Controlled Writing to NVM.* Data eviction to NVM is performed in the background by dedicated evicting threads. This approach allows for better control over concurrency, which is particularly friendly to NVM, considering its limited concurrency constraints. It also prevents an excessive number of concurrent accesses to NVM, which could impact DRAM bandwidth ($\mathcal{P}_5$).

In the following, we investigate how to effectively and efficiently build a three-tier anti-caching KV store in detail.

4 SYSTEM DESIGN

In this section, we present the overall architecture of TaC, discussing its KV operations, key components, and specific strategies applied.

4.1 Overview

4.1.1 Key-value operations. Typically, TaC is an anti-caching-based KV store, providing **set** and **get** operations based on a given key. In particular, it designates memory as its primary storage. Upon the arrival of new data, TaC initially attempts to store it in DRAM. If unsuccessful, it supplements the attempt in NVM. Generally, the expectation is to successfully locate data in memory. In situations where memory approaches full capacity, as depicted in 2(c), background evicting operations take charge of flushing cold tuples from upper tiers to lower ones. In rare cases where neither memory allocation succeeds, a **set** request fails and necessitates retrying.

When performing a **get** operation, the process is based on the data location. If the data is stored in memory, its value is directly retrieved and returned. In the case of data in SSD, an asynchronous SSD read is initiated. Besides, for tuples situated in NVM or SSD, fetching operations might be prompted according to data swapping strategies, facilitating their movement to upper tiers.

4.1.2 Book-keeping method. To identify whether a tuple resides in DRAM, NVM, or SSD, a book-keeping method is needed to maintain data location information. H-store addresses this need by employing the Evicted Table, an in-memory mapping table, to record the location of evicted tuples [11]. TaC adopts this method by incorporating the Evicted Table into the index, stored in DRAM.

Specifically, as will be introduced in Section 5, we implemented the prototype of TaC based on Memcached, which adopts a hash table as the index. Therefore, to negate the need for additional lookups in the Evicted Table, TaC incorporates tuple locations in lower tiers directly into the hash entries. When a tuple is evicted to NVM or SSD, its entry in the DRAM hash table is removed and the space is released and recycled. The key and reference information, which are essential to locate the data out of memory, are copied into a compact data structure named *meta tuple*. This meta tuple is reinserted into the hash table and used to index the tuple. This methodology ensures efficient tracking of tuple locations while reducing the need for additional lookups, thereby enhancing the overall performance of the system.

4.1.3 Challenges. The three-tier architecture introduces several challenges for an anti-caching system. Firstly, in two-tier architectures, only one data swapping path exists between memory and the anti-cache. However, the integration of NVM introduces additional paths among the three tiers. This heightened complexity emphasizes the critical need for a meticulous selection of data swapping paths, considering the distinctive characteristics, such as access granularity, of each device. In response, we employ a hybrid data swapping strategy, detailed in Section 4.2.

Secondly, opting for tuple-level data swapping in anti-caching enhances the DRAM hit rate [11, 53]. However, this approach necessitates tuple-level access tracking, which can be resource-intensive [53]. For example, widely used LRU-based eviction strategies require relocating the most recently accessed tuple to the forefront of the LRU list for each operation, potentially causing a notable performance impact. Given the inherent simplicity of KV operations in comparison to traditional database systems, the overhead associated with tuple-level access tracking may offset the benefits of data swapping. Moreover, the intricacies of the three-tier architecture demand a more precise categorization of data to seamlessly align with the hybrid data swapping strategy. In response, we adopt "Lazy LRU," an efficient multi-level access tracking mechanism, as outlined in Section 4.3.

4.2 Hybrid Data Swapping

The data-swapping strategy plays a pivotal role in relocating cold and hot tuples within TaC. In line with the characteristic of anti-caching, TaC applies tuple-level eviction across all storage tiers to evict the coldest tuples to lower tiers. However, distinct fetching strategies are employed for NVM and SSD.

For tuples residing in NVM, TaC leverages tuple-level fetching, prioritizing the hottest tuple in NVM for swapping into DRAM to fully exploit the byte-addressability of NVM.

In the case of tuples stored in SSD, data is physically fetched in blocks. H-store [11] offers two options for this scenario: block-level and tuple-level swapping. Block-level swapping involves swapping in all tuples within the fetched block, while tuple-level swapping only swaps the requested tuples, discarding the rest.

Firstly, unlike conventional database systems that often require all requested tuples to be swapped into memory for subsequent processing, a KV store does not necessarily have such a requirement. Performing swapping operations for all or only the requested tuples from a fetched block can be inefficient. Secondly, the three-tier architecture in TaC introduces more possibilities for data swapping [56]. Besides data paths between neighbor tiers, TaC capitalizes on the path between SSD and DRAM, adopting a hybrid strategy that carefully selects suitable requests for swapping operations. It assesses all tuples within a fetched block and selectively swaps those deemed appropriate, enabling swapping more tuples per I/O operation.

Intuitively, TaC prioritizes retrieving hot tuples into DRAM and warm tuples into NVM to accommodate the performance variations of different devices and the diversity in data hotness. However, achieving this necessitates determining the temperature of a tuple during fetching operations. TaC accomplishes this by leveraging a metric named "readCount", maintained by the access-tracking mechanism introduced in Section 4.3. Based on this metric, TaC classifies tuples into hot, warm, and cold categories, as summarized in Table 2. When fetching a block from SSD, TaC identifies "Hot" tuples within it and swaps them into DRAM, while "Warm" tuples are swapped into NVM.

4.3 Lazy LRU for Multi-Level Data Classification

In anti-caching systems, the cooperation between access tracking and eviction strategies is important to guide data swapping processes. In the context of a three-tier architecture, their role extends to precisely categorizing data into hot, warm, and cold, which poses unique challenges beyond the traditional hot and cold data dichotomy. In this subsection, we introduce "Lazy LRU",

Table 2: Temperature classes of tuples.

$t.readCount$	Temperature	Action
$> THR_{hot}$	Hot	Fetch from SSD or NVM to DRAM; Move atop the LRU list if in DRAM.
$> THR_{warm}$	Warm	Fetch from SSD to NVM; Move atop the LRU list if in NVM.
Others	Cold	Become ready for eviction.

a lightweight tuple-level access-tracking mechanism, which achieves multi-level data classification.

In practice, various approximation eviction strategies have been proposed to mitigate the overhead of fine-grained access tracking, including approximate LRU (ALRU)[40] and sampling-based eviction methods[11, 38]. However, neither of them is suitable for classifying data into multiple categories. Besides, these approximation methods have their limitations. For example, ALRU, instead of maintaining full LRU lists, tracks only the latest access time of each tuple and randomly selects a few tuples for eviction, potentially leading to increased eviction costs as more tuples are examined. On the other hand, the sampling method updates the LRU list only once every 'n' operation, compromising accuracy.

In TaC, we employ an alternative approximate LRU strategy named as "Lazy LRU". Lazy LRU eliminates the need to maintain LRU lists in the critical path of *get*/*set* operations, thereby avoiding potential performance bottlenecks. Instead, it captures access information during each request. To manage LRU lists efficiently, we utilize two separate background threads, one for DRAM and another for NVM, to perform batch updates.

Specifically, Lazy LRU leverages historical access information to predict the future access frequency of tuples. It categorizes tuples into distinct temperature levels based on the recorded access information as defined in Table 2. This classification allows for efficient identification of the temperature level of a tuple at any given time. This not only enables multi-level data classification but also facilitates the asynchronous updating of the LRU list as it can determine whether a tuple should be prioritized on the LRU list.

In particular, the Lazy LRU mechanism maintains two pieces of information in the metadata of a tuple:

- *time*, which is the previous access time,
- *readCount*, which records the recent access frequency.

They are both updated upon data access. Specifically, the *readCount* for a tuple t is calculated as following:

$$t.readCount = \frac{t.readCount}{currentT - t.time + 1} + 1. \quad (1)$$

in which *currentT* denotes the current time of a global clock, and it is incremented at every second.

Equation 1 draws inspiration from the cache replacement strategy known as α-Aging [35], which explicitly utilizes age information to estimate recent access frequency. In this equation, the access age of a tuple is calculated as $currentT - t.time + 1$, and the *readCount* attribute is updated based on both its access age and the previous value. In particular, TaC places significant emphasis on recent access history. To ensure that *readCount* does not become excessively large and to mitigate the impact of sudden increases in access, we impose an upper bound on its value.

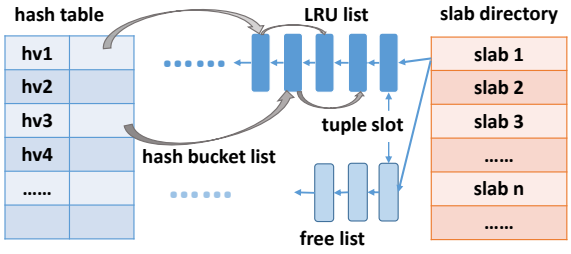


Figure 3: Data management in Memcached.

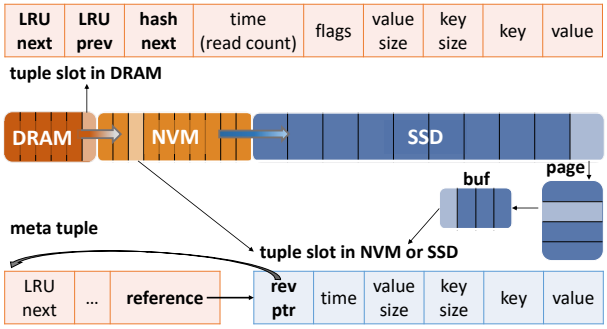


Figure 4: Data arrangement across DRAM, NVM, and SSD.

This bound is set to $2 \times THR_{hot}$, which is twice the limit for the "Hot" category.

Specifically, THR_{hot} and THR_{warm} in Table 2 represent the thresholds for "Hot" and "Warm" tuples, respectively, which are dynamically calculated. TaC periodically and randomly samples some tuples[1] to count their *readCounts*. Utilizing the histogram and the ratios of DRAM (or NVM) capacities to the total data sizes, TaC calculates the ideal thresholds for tuples suitable for DRAM and NVM. For example, when the ratio of DRAM capacity to dataset size is set at 10%, THR_{hot} will be the value at which no more than 10% of tuples have a *readCount* larger than THR_{hot}.

5 IMPLEMENTATION

We implemented TaC based on the widely used in-memory KV store, Memcached [30]. In this section, we provide an overview of the detailed implementation of TaC. First, we introduce Memcached briefly in Section 5.1, followed by a comprehensive overview of the implementation of TaC. Overall, TaC primarily incorporates the following important techniques:

- TaC manages NVM space at tuple-level like DRAM while maintaining the metadata in DRAM to mitigate small and random writes on NVM, addressing issues related to read-write asymmetry ($\mathcal{P}_2$) and small/random access overhead ($\mathcal{P}_3$) (Section 5.2).
- TaC decouples the NVM writes from the front-end read requests trying to avoid highly concurrent accesses on NVM ($\mathcal{P}_4$ and $\mathcal{P}_5$) (Section 5.3).
- TaC utilizes reference information in the metadata to filter out invalid tuples in SSD, without having to check the index thereby optimizing data retrieval (Section 5.4 and 5.6).
- TaC enhances the scalability of Memcached's memory management through partitioning techniques (Section 5.5).

5.1 Overview of Memcached

Memcached [30] is a popular open-sourced in-memory KV store, which supports *set* and *get* APIs for storing or retrieving a tuple based on a given key. As Figure 3 shows, it uses a hash table as the index and a slab directory as the memory manager to organize data. In this section, we provide a brief overview of them.

In Memcached, tuples of similar sizes are grouped into a *slab*, managed by a pre-configured slab directory as shown in Figure 3. Each slab manages an LRU list of tuples and a free list of available tuple slots. When a new tuple arrives, the system first tries to allocate a tuple slot from the free list. If unavailable, the slab requests a new page. If this request fails, the coldest tuple in the LRU list is evicted. Contrasting this, TaC retains evicted tuples in lower tiers instead of discarding them. Successful page allocation results in dividing the page into fixed-size tuple slots, as defined in the slab directory. These tuple slots are subsequently added to the free list. A background process optimizes free space across all slabs by reallocating pages from slabs with surplus free lists to those deficient in tuple slots.

Memcached utilizes a hash table to index tuples and organizes them into linked lists based on their hash values. To locate a tuple, the worker traverses the list corresponding to the hash value of the given key. Updates to existing tuples are handled via out-of-place writes. If successful, the new version is inserted into the hash table, and the old tuple slot is freed and added back to free lists. In TaC, we retain the hash index of Memcached but revamp its data storage.

A tuple slot in Memcached comprises three components: metadata, key, and value, as indicated in Figure 4 labeled by "tuple slot in DRAM". The metadata includes link pointers, sizes, and flag information. Specifically, it contains *Next* and *Prev* pointers, linking the tuple slot to its adjacent nodes in the LRU list. If a tuple slot is freed, these pointers will be used to manage the free list. The *flags* attribute within the metadata indicates whether the tuple slot is in use. An additional *HashNext* pointer in the metadata organizes the hash list in the hash bucket. To prevent write conflicts, exclusive locks are deployed to safeguard each LRU list, free list, and hash bucket in Memcached.

5.2 Data Arrangement

TaC employs different strategies to manage the space in its three storage tiers. Data space in DRAM and NVM is organized using tuple slots and slabs, akin to Memcached. Specifically, slot sizes are aligned to 256 bytes in NVM to match the access granularity of NVM [18]. In SSD, data space is divided into fixed-sized pages, which serve as the unit for space allocation and reclamation. These pages are further divided into fixed-sized *buf*s, which act as the units for read and write operations and can contain dozens to hundreds of tuples. Notably, *buf*s are not aligned with the slabs and may contain tuples from different slabs. This design choice stems from TaC's approach of not recycling tuple slots within a *buf* or page while recycling within slabs is essential for optimizing the limited space available in DRAM and NVM.

TaC adopts the format in Memcached for tuples stored in DRAM but utilizes a distinct format for those in NVM or SSD, as depicted in Figure 4. In TaC, all tuples are indexed in DRAM. Therefore, tuples in NVM or SSD omit link information in the metadata and instead include an additional field called *revPtr*, which serves as a reverse pointer to the corresponding meta tuple in DRAM. In contrast, the meta tuple in DRAM, indexed in

[1]It's important to note that the sampling target includes all stored tuples, regardless of client requests. Therefore, the statistical histogram can be regarded as representative of the overall distribution of *readCounts*.

the hash index, contains link information and references to its physical tuple in NVM or SSD. For tuples in NVM, the reference is an 8-byte pointer, while for those residing in SSD, the reference comprises a 2-byte page version, a 2-byte page identifier, and a 4-byte inner offset. This combination determines the precise physical location of the tuple.

Storing the entire index in DRAM yields substantial performance benefits. Indexes are frequently accessed components of a KV store and have a small size with random access patterns. Given that fine-grained random data accesses to NVM are significantly slower than to DRAM [48], NVM is unsuitable for indexing. Besides, TaC also keeps tuple metadata in indexes, which is crucial for *set* requests that update multiple metadata components.

In particular, the access information needed by Lazy LRU resides in the metadata of a tuple. While Memcached originally stores the access time of a tuple in 4 bytes, TaC further divides it into *time* (3 bytes) and *readCount* (1 byte), as illustrated in Figure 4. Specifically, *time* increments at the second level, making 3 bytes sufficient for data temperature identification.

5.3 Data Eviction

In TaC, the responsibility for managing data movement operations, such as data eviction and data fetching, is delegated to dedicated threads. This delegation of tasks helps optimize the utilization of worker threads, allowing them to focus on serving requests more efficiently [11]. In particular, with dedicated threads responsible for data flushing to NVM, the NVM writes are decoupled from front-end requests and alleviate the influence of limited NVM write concurrency on the system performance.

Specifically, when the free space in memory reaches a watermark (4 MB for DRAM and 256 MB for NVM in our evaluation), the eviction process is triggered. This process entails transferring cold tuples from higher (and faster) storage tiers to lower (and slower) tiers. This ensures the efficient utilization of storage resources and maintains system performance.

During eviction, TaC relies on the Lazy LRU mechanism to determine which tuples to evict. Background threads for Lazy LRU move hot or warm tuples atop the LRU lists of DRAM or NVM respectively, while eviction threads select tuples from the bottom of LRU lists for eviction. Typically, when a tuple is evicted to NVM, it is written directly to a tuple slot in NVM. However, if tuples are evicted to SSD, they are initially packed into *buf*s, which are subsequently flushed to SSD when they reach the *buf* capacity.

5.4 Data Fetching

Similar to the data eviction process, TaC employs dedicated threads to asynchronously select and fetch hot tuples from lower tiers to upper tiers. When accessing tuples in NVM, "Hot" tuples will be recorded in a thread-local queue. Thereby, fetching threads will check these queues and fetch hot tuples in NVM into DRAM.

When accessing SSD, once the accessed tuple is warm or hot, TaC will record the *buf* it residing in and notify the fetching threads. The fetching threads then regard all tuples in the *buf* as candidates for swapping. However, as updates are performed out-of-place in TaC, candidate tuples may become outdated. Initially, to verify the validity of a tuple, its key can be used to traverse the hash table to determine if the tuple is in use or not. However, traversing the index needs to fetch the lock of related hash buckets. To avoid this, TaC leverages the *revPtr* in the SSD-resident

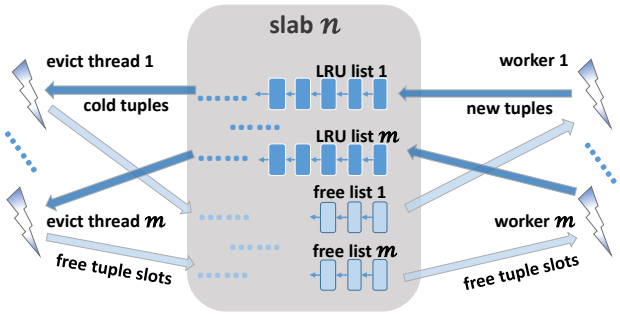


Figure 5: Illustration of partitioned memory management.

tuple as depicted in Figure 4, without checking the index. In particular, an SSD-resident tuple can be considered as valid if and only if it can satisfy three conditions:

- its *revPtr* is not NULL and the relative meta tuple is in use (indicated by *flags*);
- its key remains the same as the one in the meta tuple;
- its meta tuple refers to itself.

After confirming the validity of a candidate tuple, TaC will check its temperature to determine whether to fetch it and where to fetch it. According to Table 2, "Hot" tuples are fetched into the DRAM, "Warm" tuples into the NVM, and "Cold" tuples will stay in the SSD.

5.5 Scalable Memory Management

As discussed in Section 5.1, Memcached manages tuples by slabs. Each slab maintains an LRU list for data eviction and a free list for tuple allocation. However, these lists are protected by exclusive locks, which can easily become a hot spot of contention. The introduction of data evicting and fetching threads of the anti-caching architecture has made the problem even more severe. To address this issue, TaC partitions memory management tasks such as data eviction, data fetching, and LRU maintenance. In particular, Figure 5 illustrates how the data eviction tasks are partitioned, while data fetching and LRU maintenance can be handled in similar ways.

As illustrated in Figure 5, TaC divides both the LRU list and free list of a slab into m partitions, with each partition assigned to a corresponding evicting thread responsible for data eviction and space recycling. Each worker is also assigned to a particular partition, allowing it to operate on its designated free-list and LRU-list partition when inserting or updating a tuple. This partitioning scheme helps reduce contention on the exclusive locks associated with each list and improves overall concurrency.

Increasing the number of partitions can help eliminate contention, but it can also result in excessive space and performance overhead and potentially impact the accuracy of LRU. To address this, TaC sets the number of partitions to the maximum number of threads initialized by the system, providing enough parallelism without excessive overhead. Load balancing is also a crucial factor to consider when utilizing the partitioning approach. TaC addresses this by enabling threads to steal work from other partitions when they're starved. Due to space limitations, we omit the details.

5.6 Space Reclamation

Following the original Memcached, TaC employs out-of-place updates for all *set* requests, which allows changing value sizes when

updating a tuple. When a tuple in DRAM or NVM is updated, its original slot is immediately recycled to the free list. However, for SSD, TaC reclaims space in pages, meaning that a page can only be reclaimed when all of its tuples are outdated/invalid. This approach can cause space fragmentation and potentially exhaust SSD capacity if there are too many out-of-place updates. (Note that a tuple in SSD could become invalid if (1) the tuple has been updated or deleted or (2) it has been swapped into DRAM or NVM.) This may cause space fragmentation.

To address this issue, TaC proactively performs defragmentation when the SSD is about to be exhausted. The defragmentation process starts from the oldest page, reads in all valid *buf*s, and re-evicts their valid tuples to new pages while filtering out invalid tuples according to the conditions introduced in Section 5.4. When tuples are moved to new locations of SSD, only the reference information in the meta tuples needs to be updated while their corresponding meta tuples in DRAM remain intact.

5.7 About Failure Recovery

Data loss can be tolerated in some specific circumstances, such as being used as the application-level cache [31], where the lost data can be refilled by following requests. However, the need for failure recovery is critical in many other contexts. We posit that the design of TaC can be readily adapted to support failure recovery and introduce our solution briefly below.

Essentially, an operation log in NVM can assist in data recovery in the event of a system crash. This operation log offers several advantages. Firstly, due to its sequential writing nature, NVM can leverage its near-DRAM performance. Secondly, since data in NVM and SSD is already persistent, it can serve as a checkpoint and be used to truncate the log. This means that only the data in DRAM needs to be backed up, which is typically the smallest portion.

To ensure the integrity of persisted data, checksums can be added to tuple slots in NVM or SSD. Additionally, if the old version of a tuple in SSD has not been reclaimed, the *time* attribute of tuples in NVM and SSD, as illustrated in Figure 4, can be used to identify the latest version. This is because the *time* attribute records the last access time before it is flushed, and an older version will not be accessed again after being updated. In other words, the *time* attribute of the new tuple, which is no less than its creation time, is certainly larger than that of the old version.

6 EVALUATION

6.1 Experimental Environment

We conducted experiments on a dual-socket machine managed by Linux kernel 5.4.0-126. Each socket was equipped with an Intel® Xeon® Gold 6326 CPU (2.90GHz) with 16 physical cores. Each physical core had a 48KB L1 data cache, a 32KB L1 instruction cache, and a 1.25MB L2 cache, and can be forked into 2 logical ones by hyper-threading. A 24MB L3 cache was shared by all physical cores in the same socket. The machine was equipped with 128GB DRAM, 1TB NVM (Intel® Optane™ Persistent Memory 200 Series) per socket, and 1.92TB SAMSUNG SSD (MZQL21T9HCJR-00A07 series). All NVM DIMMs were configured to App Direct mode, exposing NVM and DRAM to programmers as two separate memory tiers and allowing explicit control of the utilization of different kinds of memory. All experiments were performed on a single socket with all memory (including DRAM and NVM) accesses and CPU utilization being restricted to the same socket to rule out potential NUMA impacts.

6.1.1 Workload. We conducted our evaluation using the YCSB benchmark [9]. The benchmark primarily comprises concurrent *get* and *set* requests. To assess TaC in different scenarios, we tried three representative workloads as follows:

- Read-Only (**YSCB-RO**): 100% reads
- Read-Heavy (**YCSB-RH**): 95% reads and 5% updates
- Write-Heavy (**YCSB-WH**): 50% reads and 50% updates

In particular, the workloads were configured to follow the Zipfian distribution, with a default factor of 0.99. Each tuple consisted of an 8-byte key and a 1000-byte value. For each experiment, we preloaded 50 million tuples (over 50GB) unless otherwise specified. We measured system throughput as our primary evaluation metric, as in previous studies [5, 11, 56]. All reported results are the averages of 10 consecutive runs, each lasting 30 seconds.

6.1.2 Systems for Comparison. We mainly evaluated TaC against the following alternative designs:

- *Anti-NVM*, a two-tier design that utilizes NVM as the primary storage, supplemented by SSD, as illustrated in Figure 2(a). In this design, DRAM is the cache of NVM. In order to better control the allocation of DRAM and NVM, we implemented a variant based on Memcached using the "App Direct" Mode of Optane. When a tuple is updated, the new data is initially written to the cache in DRAM and subsequently flushed to NVM in the process of cache eviction.
- *Anti-2*, a two-tier design that does not differentiate between DRAM and NVM, as illustrated in Figure 2(b). It is also implemented based on Memcached. In particular, without making a distinction between DRAM and NVM, it brings all “Warm” data, including those categorized as “Hot” in Table 2, into memory.
- *FASTER-NVM*, another variant of the design in Figure 2(b) which is implemented on top of a two-tier anti-caching KV store, FASTER [5]. It treats NVM as DRAM to extend the memory part of FASTER while retaining SSD as a supplement, adhering to its original design.
- *Spitfire*, which is a three-tier caching architecture, treating DRAM and NVM as buffers for SSD [56].
- *Prism*, which is a KV store on heterogeneous storage devices [42], using DRAM for read cache and NVM for write buffer and index.

In the experiments, for a fair comparison, all systems were allocated an equal amount of memory and bound to identical CPU resources. Specifically, all Memcached variants adopted the same in-memory indexing and bookkeeping methods as TaC, while FASTER-NVM and Spitfire received the same amount of DRAM allocation as Memcached. By default, TaC employed 8 evicting threads, with the number of fetching threads equal to the number of workers. The impact of these settings on the performance of TaC is discussed in Section 6.4. Additionally, the number of partitions was set to 128, slightly exceeding the total count of workers and background threads, as explained in Section 5.5.

6.2 Performance Comparison and Analysis

To evaluate the performance of different designs, we tested all comparison systems on a variety of circumstances.

6.2.1 Experiments on varying degree of parallelism. Figure 6 illustrates the impact of varying the number of workers on the throughput of subject systems, with fixed DRAM and NVM sizes

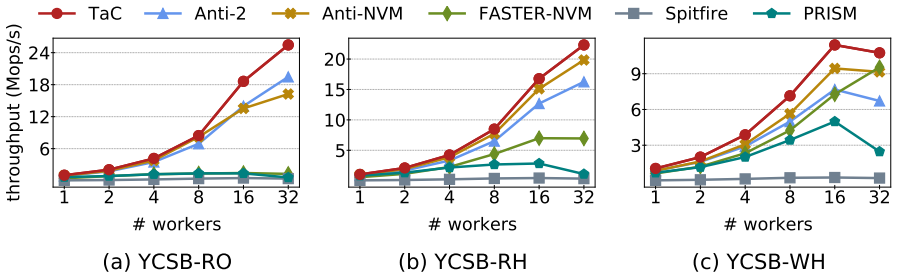


Figure 6: Impact of the degree of concurrency on throughput.

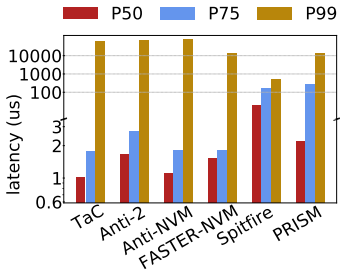


Figure 7: Read latency on YCSB-WH.

Table 3: Memory hit rates on the YCSB-RH workload.

	TaC	Anti-2	Anti-NVM
DRAM	61.4%	9.1%	59.8%
NVM	21.0%	76.3%	18.8%

of 4GB and 32GB, respectively. Most systems exhibited efficient scaling on the multi-core machine, with all Memcached variants effectively scaling up to 16 workers, matching the number of physical cores. However, when the number of workers exceeded this threshold, contention on lock resources and SSD I/Os among workers and background threads increased significantly, particularly for write-heavy workloads. This resulted in a plateau or a slight decrease in throughput. For a deeper insight into performance differences between the systems, we present the read latency of all systems in Figure 7 and the memory hit rates (i.e., the ratios of read requests fetching data from DRAM or NVM) of the Memcached variants in Table 3. Generally, the results verify the superiority of TaC over alternative designs. Moreover, several key observations can be made:

(a) The anti-caching architecture outperforms the caching architecture regarding the throughput.

As depicted in Figure 6, anti-caching-based variants outperformed Spitfire and Prism, which utilize the caching or buffering architecture. This is because the former was originally designed for in-memory systems, featuring more efficient in-DRAM indexes and finer-grained data swapping compared to Spitfire and Prism, which are inherently disk-resident systems. Notably, Figure 7 reveals that Spitfire and Prism experienced higher P50 and P75 latency than others. However, Spitfire exhibited a smaller P99 latency due to asynchronous SSD reads in other systems, which can result in occasional higher latency, especially when conflicting with writes.

In particular, Prism employs an in-NVM index and asynchronous SSD read to optimize system performance. Further, it employs DRAM as a read cache for hot data in SSD to optimize read performance and NVM as a write buffer to enhance write operations. However, the separation prevents it from leveraging NVM in read-intensive workloads. Moreover, without data transformation between DRAM and NVM, it struggles to handle hot data in NVM.

Besides, there is a significant discrepancy between our results for Spitfire and those reported in the original paper [56]. This is because the original paper measured the throughput of the buffer manager and a single *set* or *get* KV operation would result in multiple calls to the buffer manager. Additionally, caching architecture naturally supports data persistence and is more suited for failure recovery scenarios. Therefore, the choice between these architectures may vary based on specific application requirements.

(b) Distinguishing between NVM and DRAM enhances system performance by improving DRAM hit rates and NVM accessing efficiency.

As illustrated in Figure 6, TaC consistently outperforms FASTER-NVM and Anti-2, both of which employ two-tier anti-caching architectures without distinguishing between DRAM and NVM. Notably, when compared to Anti-2, TaC exhibits remarkable performance improvements, exceeding 1.48x for the YCSB-WH workloads.

This superiority comes from the ability of TaC to differentiate between hot and warm data, with hot data stored in DRAM and warm data in NVM, resulting in an improved DRAM hit rate. In contrast, Anti-2 treats NVM similarly to DRAM, resulting in the storage of hot data in both. Specifically, for read-heavy workloads, TaC achieves a DRAM hit rate of 61.4%, while Anti-2 lags at 9.1%. Given the inferior performance of NVM compared to DRAM, this negatively impacts the performance of Anti-2. Besides, the performance gap widens further with write-heavy workloads since NVM experiences more pronounced performance degradation for writes compared to reads. This issue becomes particularly severe during metadata maintenance in the process of *set* operations, which involves numerous small and random memory accesses.

FASTER-NVM performs better on write-heavy workloads because it directs all write requests to memory, capitalizing on its latch-free in-DRAM hash index and in-place updates strategy. However, similar to Anti-2, it suffers from a low DRAM hit rate. Notably, FASTER-NVM struggles significantly with read-only workloads, primarily because FASTER only brings newly written data into memory storage, neglecting optimization for read requests. Consequently, it experiences exceptionally low memory hit rates in read-only scenarios.

(c) The three-tier architecture outperforms the "Memory Mode" architecture due to better memory utilization and reduced data synchronization cost.

Figure 6 also demonstrates that TaC outperforms Anti-NVM, which utilizes DRAM as a cache of NVM. This performance discrepancy can be attributed to two primary factors: Firstly, Anti-NVM exhibits data redundancy between NVM and DRAM, resulting in inefficient memory utilization and a lower memory hit rate compared to TaC. This effect is particularly pronounced in read-only workloads. Secondly, Anti-NVM incurs maintenance overhead due to data synchronization between the DRAM cache and the primary data in NVM, primarily for write requests. In the case of TaC, hot tuples can be stored directly in DRAM tuple slots without necessitating NVM involvement. Conversely, in the design of Anti-NVM, space in NVM must be allocated and

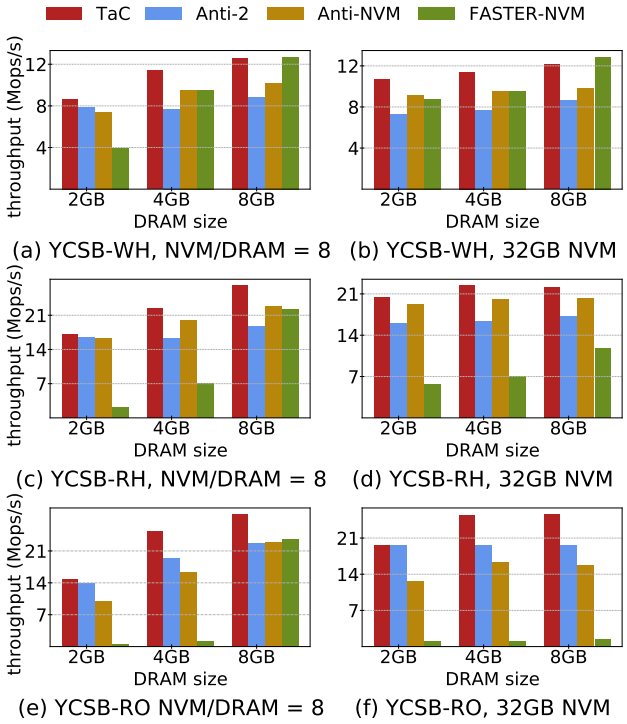


Figure 8: Impact of DRAM and NVM sizes on throughput.

managed, leading to additional overhead for maintaining the two copies in both DRAM and NVM.

In the following experiments, we applied 32 workers for read-only and read-heavy workloads and 16 workers for the write-heavy workloads except for the FASTER-NVM, which applied 32 workers for write-heavy workloads to get its top performance.

6.2.2 Experiments on varying DRAM and NVM sizes. The performance of anti-caching systems is usually heavily influenced by the capacity of DRAM and NVM. To assess this, we conducted a series of experiments, varying the sizes of DRAM and NVM across all systems. Initially, we held the DRAM/NVM size ratio at 8 while adjusting the size of DRAM, and then held the NVM capacity at 32GB while adjusting the size of DRAM, yielding different DRAM/NVM size ratios. As Figure 8 illustrates, TaC outperforms other systems across a wide range of memory configurations. It also leads to the following observations:

(a) Increased size of memory tends to soften the difference between TaC and Anti-NVM on read operations.

As illustrated in Figure 8, the performance of Anti-NVM exhibits significant improvement as memory size increases. To provide a concrete example, when using 2GB of DRAM and 16GB of NVM, TaC outperforms Anti-NVM by 1.50x in a read-only workload. However, this performance advantage diminishes to 1.27x when using 8GB of DRAM and 64GB of NVM. With larger NVM capacities, both TaC and Anti-NVM allocate most of the data in NVM, reducing the impact of wasted memory space. Moreover, a larger DRAM allows for a greater amount of hot data to be stored in DRAM, significantly reducing the overhead associated with cache eviction. Nevertheless, even when equipped with 64GB of NVM, TaC maintains its performance superiority over Anti-NVM, particularly in write-heavy workloads. This continued advantage comes from inherent challenges in Anti-NVM regarding the data synchronization between the two copies residing in DRAM and NVM.

(b) Increasing the memory size would enlarge the superiority of TaC over Anti-2 while an increased DRAM/NVM size ratio would soften its superiority on write operations.

As indicated in Figure 8(c), when employing 2GB of DRAM and 16GB of NVM, the throughput of TaC is 1.04x that of Anti-2, but this difference increases to 1.44x when using 8GB of DRAM and 64GB of NVM. This outcome aligns with expectations: when more data can be accommodated in memory (whether DRAM or NVM), overall performance becomes predominantly influenced by the efficiency of DRAM utilization. Consequently, the superior performance of TaC is attributed to its ability to enhance the DRAM hit rate. However, as the ratio of DRAM to total memory sizes increases, there are more requests served by DRAM, diminishing the performance gap between Anti-2 and TaC, especially on write operations. To illustrate, when employing 32GB of NVM and 2GB of DRAM, the DRAM hit rate of Anti-2 for write-heavy workloads is only 9.1%, but it increases to 24.5% when using 8GB of DRAM.

Notably, FASTER-NVM exhibits outstanding performance in the configuration featuring 8GB of DRAM and 64GB of NVM. This exceptional performance is attributed to the ability of memory to hold all data with such a configuration, enabling the latch-free index of FASTER to fully capitalize on its strengths.

6.2.3 Experiments on varying dataset sizes. In this part, we conducted additional experiments to investigate the impact of varying dataset sizes on the performance of different anti-caching systems. In these experiments, we held the sizes of DRAM and NVM constant at 20GB and 80GB, while varying the dataset size. The results are presented in Figure 9. Additionally, the figure provides the performance of Plush [44] and Halo [15], which are state-of-the-art hybrid DRAM and NVM Key-Value stores, as references. In particular, Plush adopts an LSM-tree-like architecture and we configured its DRAM tier as 32GB with other configurations as default settings. Basically, we could get the following key observations:

(a) Applying NVM directly to well-optimized DRAM-oriented systems can result in significant performance degradation.

Notably, FASTER-NVM outperforms Memcached variants when the entire dataset fits into memory (i.e., no larger than 100GB), particularly when most of the data can reside in DRAM (i.e., around 20GB). This advantage arises from the highly efficient in-DRAM index of FASTER and its ability to capitalize on the in-place update strategy without the engagement of SSD. However, as the dataset size increases and more data is stored in NVM, its performance declines substantially. For instance, Figure 9(a) shows that its throughput for write-heavy workloads decreases by 43% as the data size increases from 25GB to 50GB. This drop in performance can be attributed to its DRAM-oriented update strategy, which leads to suboptimal performance on NVM. A similar phenomenon is observed with Anti-2. In contrast, TaC exhibits a more stable performance as the dataset size grows. This further highlights that directly applying NVM to existing, well-optimized two-tier anti-caching systems is not a favorable approach.

(b) The performance of anti-caching-based systems is competitive to that of hybrid DRAM and NVM systems when the data could fit in memory while providing larger capacity.

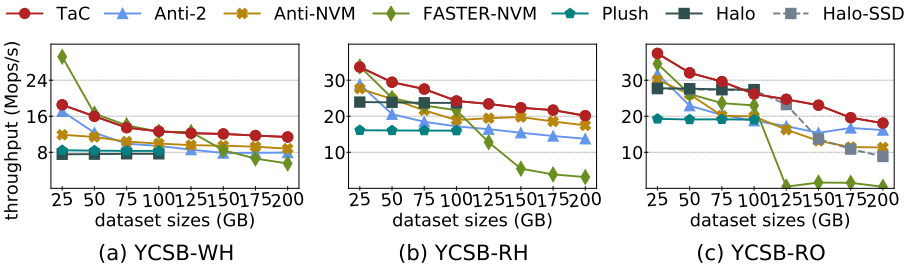


Figure 9: Impact of dataset sizes on throughput.

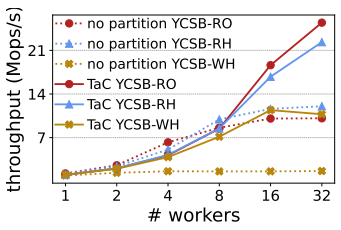


Figure 10: Impact of partitioning.

Plush [44] and Halo [15] stand as representative hybrid DRAM and NVM KV stores. As indicated in the figure, Plush excels in write-heavy workloads, leveraging its DRAM tier as an effective write buffer. Conversely, Halo showcases superior performance in read-intensive scenarios, capitalizing on its optimized in-DRAM hash index and pre-fetching-optimized read strategy.

When the dataset fits into memory, TaC demonstrates competitive performance compared to Plush and Halo, as all data in TaC is stored in memory. In particular, in scenarios where the majority of the data can reside in DRAM, TaC outperforms both Plush and Halo significantly across various workloads. This is credited to the hotness-aware design of TaC, enhancing DRAM utilization and ensuring that the majority of requests are serviced by DRAM.

Specifically, with write-heavy workloads, TaC outperforms both Plush and Halo. This is attributed to the anti-caching design which maximizes the utilization of DRAM for serving write requests. In contrast, despite Halo leveraging a pre-core buffer to optimize small write operations, it still necessitates all write requests to reach NVM. Similarly, the append-only update strategy in Plush, along with its extra payload log for values larger than 8 bytes, also mandates that all write requests would reach NVM once the DRAM is full. In contrast, TaC facilitates the timely reclamation and recycling of old tuple slots in DRAM, resulting in fewer NVM writes.

As the dataset size surpassed the memory capacity, Plush and Halo encountered challenges as they assumed that everything could be accommodated in NVM. In contrast, the anti-caching architecture is purposefully designed to handle such scenarios, while providing promising performance, as illustrated in the figure.

To illustrate the impact of integrating SSD into the system, we introduced a variant of Halo, named "Halo-SSD". In Halo-SSD, data storage is divided into two parts: the first part remains in NVM, consistent with the original design, occupying 100GB; the second part is allocated to SSD. When reading a tuple located in the second part, it incurs an SSD read instead of the original NVM read. In Figure 9(c), the performance of Halo-SSD on read-only workloads is presented. As evident, the introduction of SSD leads to a sharp drop in the read performance of Halo-SSD, highlighting the non-trivial nature of extending the hybrid DRAM and NVM design to include SSD. The primary challenges revolve around identifying data temperatures and fully leveraging memory performance, both of which TaC is dedicated to resolving. In conclusion, the three-tier anti-caching design proves to be an effective choice when data size exceeds memory capacity.

6.3 Optimization Impact Analysis

In this subsection, we performed studies on the implementation decisions of TaC, including the partitioned memory management, the hybrid data swapping strategy, and the Lazy LRU strategy.

6.3.1 Partitioned memory management. Partitioned memory management indeed plays a significant role in ensuring the scalability of TaC. Figure 10 exhibits the multi-core scalability of TaC against that of the original Memcached enhanced with anti-caching (termed *no partition*). From the figure, it is evident that with partitioned memory management, the system can scale significantly better, particularly on write-heavy workloads, where the original Memcached has little scalability after equipping with the anti-cache. This is because the write-heavy workload results in more data eviction operations, leading to more contention on LRU lists and free lists. As a result, the impact of partitioning in reducing their lock contention becomes more noticeable. Partitioning the memory effectively limits the lock contention within each partition, allowing for improved parallelism and better utilization of multi-core resources.

6.3.2 Hybrid data swapping. To verify the effectiveness of the hybrid data swapping strategy, we compared TaC against another variant of Memcached, termed *Anti-Tuple*. *Anti-Tuple* adopts the same three-tier architecture as TaC but only performs tuple-level swapping. This means that when fetching data from the SSD, it only swaps in the exact tuples requested by the *get* operations. In contrast, when TaC fetches a *buf* from the SSD, it evaluates all the tuples in the buffer and selectively swaps in the hot and warm tuples. This strategy allows TaC to maximize the efficiency of SSD operations, resulting in improved performance.

As depicted in Figure 11, the hybrid data swapping strategy outperformed the pure tuple-level data swapping strategy on the read-only and read-heavy workloads. However, it has little effect on the write-heavy workloads as it does nothing for write requests and the frequent *set* requests bring updated data into memory, which also results in less influence of the data swapping strategy. The results demonstrate the advantage of the hybrid swapping strategy, which provides more efficient data access.

6.3.3 Lazy LRU. Lazy LRU plays an important role in dividing the data into different temperature levels and guiding the data swapping operation, which is hard to implement by other alternatives. Besides, it could also improve the cache hit ratios and the efficiency of systems. To further validate its efficiency, we conducted a simulation experiment similar to the one performed in FASTER [5]. In this simulation, the tuple at the bottom of the LRU list is continuously evicted to make space for new tuples. We applied various eviction strategies, including LRU, Lazy LRU, ALRU, and Sampled LRU, across a range of cache sizes.

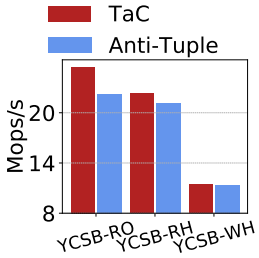


Figure 11: Throughput w.r.t. swapping strategies.

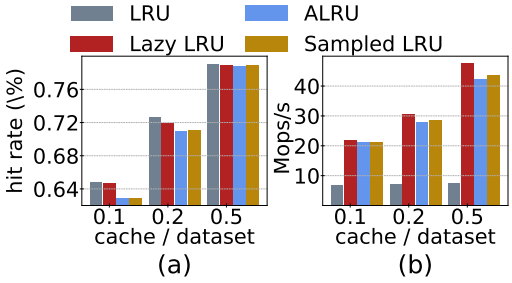


Figure 12: Cache hit rate (a) and throughput (b) w.r.t. eviction strategies.

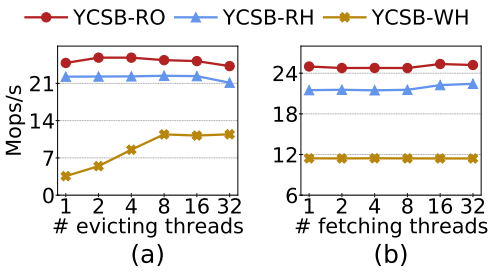


Figure 13: Throughput w.r.t. numbers of evicting (a) and fetching (b) threads.

Figure 12(a) illustrates the cache hit rates of different approaches, while Figure 12(b) plots their throughputs. In each experiment, we employed 16 workers and set the Zipfian factor to 0.99. The results showed that Lazy LRU outperformed alternatives in both throughput and cache hit rate.

The figure clearly shows that the original LRU suffers from very low throughput, as it requires updating the LRU list for almost every read operation. ALRU performs fewer updates on the LRU lists, but this comes at the cost of needing to access more tuples when performing eviction. Sampled LRU misses a significant amount of information about data accesses, which negatively impacts its accuracy in approximating LRU. In contrast, Lazy LRU, which only delays updates on the LRU lists, has more information available for approximating LRU. It also minimizes interference on the workers processing user requests, enhancing the throughput. Thus, the Lazy LRU strategy demonstrates superior performance in managing cache eviction and data access tracking.

6.4 Parameter Sensitive Analysis

In this part, we performed studies on the influence of the number of evicting and fetching threads. The results are shown in Figure 13.

The number of evicting threads can have a significant impact on data eviction operations. Specifically, too few evicting threads can cause eviction lag, which can have a negative impact on write performance. However, too many evicting threads can lead to high contention on locks and SSD I/O operations, which can also be slightly detrimental to performance, especially for read-intensive workloads. Overall, 8 threads, half the number of physical cores, seem to yield good performance across various scenarios.

The number of fetching threads shows little influence on the systems. Firstly, for write-heavy workloads, the set requests would bring hot data into DRAM resulting in less requirement on data swapping operations. Secondly, the skewness of the requests also helps restrict the amount of data swapped due to the limited amount of hot data. However, more fetching threads allow hot data to be migrated into upper tiers quicker, resulting in a slightly better read performance. Therefore, we suggest the same number of fetching threads as the workers.

7 CONCLUSION

This paper discussed several potential designs for integrating NVM into anti-caching KV stores. Specifically, considering the characteristics of NVM, we present an effective three-tier anti-caching design, TaC. It addresses the challenges of managing data across three different storage tiers by introducing a hybrid data swapping strategy and a lightweight access tracking mechanism implementing multi-level data classification. Through extensive experimentation, we demonstrated the superiority of TaC over alternative designs. The results indicate the three-tier design is the most efficient and effective method for incorporating NVM into anti-caching KV stores.

As for future work, there have been other kinds of memories that gained a lot of attention, such as CXL memory and High Bandwidth Memory (HBM). Firstly, the specific performance characteristics of alternative devices, such as CXL memory, require more experimental exploration and targeted studies. Secondly, expanding and generalizing the anti-caching architecture to be a modular framework that enables seamless incorporation of different devices remains a valuable topic. Besides, the extended persistence domain offered by the Optane Series 200 also introduces additional optimization chances regarding the persistence of anti-caching architectures.

8 ACKNOWLEDGE

This project is supported by the NSFC Project (China) No. 92270202. It is also partially supported by a grant funded by the Ministry of Education (Singapore) (Title: inPMdb: An in-Persistent Memory Database System; WBS NO: A8000082-00-00) and Shanghai Engineering Research Center of Big Data Management. Moreover, we appreciate the valuable suggestions from anonymous reviewers.

REFERENCES

[1] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (2013), 13:1–13:35. https://doi.org/10.1145/2463585.2463589

[2] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2023. WiscSort: External Sorting For Byte-Addressable Storage. *Proc. VLDB Endow.* 16, 9 (2023), 2103–2116. https://www.vldb.org/pvldb/vol16/p2103-banakar.pdf

[3] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (2021), 1544–1556. https://doi.org/10.14778/3461535.3461543

[4] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-Bench: Benchmarking Persistent Memory Access. *Proc. VLDB Endow.* 15, 11 (2022), 2463–2476.

[5] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 275–290. https://doi.org/10.1145/3183713.3196898

[6] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020,*

James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 1077–1091. https://doi.org/10.1145/3373376.3378515
[7] CXL Consortium. 2022. Compute Express Link (CXL) Specification. https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf
[8] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (may 2023), 27 pages. https://doi.org/10.1145/3588726
[9] 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, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152
[10] Lixiao Cui, Kewen He, Yusen Li, Peng Li, Jiachen Zhang, Gang Wang, and Xiaoguang Liu. 2023. SwapKV: A Hotness Aware In-Memory Key-Value Store for Hybrid Memory Systems. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2023), 917–930. https://doi.org/10.1109/TKDE.2021.3077264
[11] Justin A. DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (2013), 1942–1953. https://doi.org/10.14778/2556549.2556575
[12] DRAM price 2024 (accessed 2024). February 2024 Server Memory Prices. https://memory.net/memory-prices/.
[13] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (2020), 626–639. https://doi.org/10.14778/3436905.3436921
[14] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two. *Proc. VLDB Endow.* 15, 11 (2022), 2477–2490. https://www.vldb.org/pvldb/vol15/p2477-wang.pdf
[15] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1049–1063. https://doi.org/10.1145/3514221.3517884
[16] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (2021), 785–798. https://doi.org/10.14778/3446095.3446101
[17] Kaisong Huang, Yuliang He, and Tianzheng Wang. 2022. The Past, Present and Future of Indexing on Persistent Memory. *Proc. VLDB Endow.* 15, 12 (2022), 3774–3777. https://www.vldb.org/pvldb/vol15/p3774-wang.pdf
[18] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. *Proc. VLDB Endow.* 16, 6 (2023), 1249–1263. https://www.vldb.org/pvldb/vol16/p1249-huang.pdf
[19] Yihe Huang, Matej Pavlovic, Virendra J. Marathe, Margo I. Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 967–979. https://www.usenix.org/conference/atc18/presentation/huang
[20] Intel. 2022. Intel Reports Second-Quarter 2022 Financial Results.
[21] JEDEC. 2021. DDR4 NVDIMM-P BUS PROTOCOL. https://www.jedec.org/system/files/docs/JESD304-4-01.pdf.
[22] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G. Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 821–837. https://www.usenix.org/conference/atc21/presentation/kassa
[23] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 185–196. https://doi.org/10.1109/ICDE.2018.00026
[24] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 447–461. https://doi.org/10.1145/3341301.3359628
[25] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (2019), 574–587. https://doi.org/10.14778/3372716.3372728
[26] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, William D. Gropp, Pete Beckman, Zhiyuan Li, and Francisco J. Cazorla (Eds.). ACM, 26:1–26:10. https://doi.org/10.1145/3079079.3079089
[27] Haikun Liu, Renshan Liu, Xiaofei Liao, Hai Jin, Bingsheng He, and Yu Zhang. 2020. Object-Level Memory Allocation and Migration in Hybrid Memory Systems. *IEEE Trans. Computers* 69, 9 (2020), 1401–1413. https://doi.org/10.1109/TC.2020.2973134
[28] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2021. Scaling Dynamic Hash Tables on Real Persistent Memory. *SIGMOD Rec.* 50, 1 (2021), 87–94. https://doi.org/10.1145/3471485.3471506
[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28. https://doi.org/10.1145/3033273
[30] Memcached 2023 (accessed September, 2023). Memcached. https://memcached.org/.
[31] Jhonny Mertz and Ingrid Nunes. 2017. A Qualitative Study of Application-Level Caching. *TOSEM* 43, 9 (2017), 798–816.
[32] NVM price 2023 (accessed September, 2023). NVM price. https://www.intel.sg/content/www/xa/en/products/details/memory-storage/optane-dc-persistent-memory.html.
[33] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 371–386. https://doi.org/10.1145/2882903.2915251
[34] Prashant Pandey, Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance Hash Tables Through Stability and Low Associativity. *Proc. ACM Manag. Data* 1, 1 (2023), 47:1–47:26. https://doi.org/10.1145/3588727
[35] Stefan Podlipnig and László Böszörményi. 2003. A survey of Web cache replacement strategies. *ACM Comput. Surv.* 35, 4 (2003), 374–398. https://doi.org/10.1145/954339.954341
[36] Madhava Krishnan Ramanathan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 773–787. https://www.usenix.org/conference/atc21/presentation/krishnan
[37] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.* 52, 4-5 (2008), 465–480. https://doi.org/10.1147/rd.524.0465
[38] 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 *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 392–407. https://doi.org/10.1145/3477132.3483550
[39] Redis 2023 (accessed September, 2023). Redis. https://redis.io/.
[40] Redis ALRU 2023 (accessed September, 2023). Redis ALRU. https://redis.io/docs/manual/eviction/.
[41] RocksDB 2023 (accessed September, 2023). RocksDB. http://rocksdb.org/.
[42] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 588–602. https://doi.org/10.1145/3575693.3575722
[43] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1541–1555. https://doi.org/10.1145/3183713.3196897
[44] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *Proc. VLDB Endow.* 15, 11 (2022), 2895–2907. https://doi.org/10.14778/3551793.3551839
[45] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, and Yuanyuan Dong. 2022. PATS: Taming Bandwidth Contention between Persistent and Dynamic Memories. In *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022, Antwerp, Belgium, March 14-23, 2022*, Cristiana Bolchini, Ingrid Verbauwhede, and Ioana Vatajelu (Eds.). IEEE, 885–890. https://doi.org/10.23919/DATE54114.2022.9774762
[46] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia
[47] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 488–505. https://doi.org/10.1145/3492321.3519556
[48] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close

look at its on-DIMM buffering. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 488–505. https://doi.org/10.1145/3492321.3519556

[49] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. *login Usenix Mag.* 45, 3 (2020). https://www.usenix.org/publications/login/fall2020/yang

[50] J. Joshua Yang and R. Stanley Williams. 2013. Memristive devices in computing system: Promises and challenges. *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (2013), 11:1–11:20. https://doi.org/10.1145/2463585.2463587

[51] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[52] Geoffrey Yu, Markos Markakis, Andreas Kipf, Per ake Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16, 1 (2022), 99–112. https://doi.org/10.14778/3561261.3561270

[53] Hao Zhang, Gang Chen, Beng Chin Ooi, Weng-Fai Wong, Shensen Wu, and Yubin Xia. 2015. "Anti-Caching"-based elastic memory management for Big Data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 1268–1279. https://doi.org/10.1109/ICDE.2015.7113375

[54] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: a key-value store for optane persistent memory. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 194–209. https://doi.org/10.1145/3447786.3456237

[55] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, Arif Merchant and Hakim Weatherspoon (Eds.). USENIX Association, 207–219. https://www.usenix.org/conference/fast19/presentation/zheng

[56] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2195–2207. https://doi.org/10.1145/3448016.3452819