A Survey on In-Memory KV Store Designs for Today's Data Centers

Showan Esmail Asyabi

Abstract

In-memory KV stores are non-persistent storage backbones of an ever-growing number of services in today's data centers. In-memory KV stores substantially reduce the latency and increase throughput compared to alternative slow database systems while reducing the traffic to those backend systems. The importance of in-memory KV stores has sparked a lot of research works over the recent years. The value of in-memory KV stores is mainly judged by their throughput, tail latency, scalability, memory efficiency, and power consumption.

This paper highlights the characteristics of workloads of in-memory KV stores in large-scale data centers such as Facebook and Twitter data centers. We discuss the shortcomings of popular production systems such as Memcached. We then discuss the state-of-the-art attempts in improving throughput, latency performance, scalability, memory efficiency, and power consumption of in-memory KV stores. Finally, we discuss open problems.

1 Introduction

In-memory KV stores such as Memcached [2] and Redis [3] are non-persistent storage backbones for an ever-growing number of large-scale applications in today's data centers such as Twitter, Facebook and Dropbox [8] [15][14] [6]. They reduce latency and increase throughput by caching objects retrieved from slow databases or storage systems [8] [11]. They mitigate heavy computations by caching the results of past computations. For example, one Facebook Memcached pool achieves a 98.2% hit rate with an average latency of 100 µs, while access to MySQL takes 10 ms, supporting data retrieval more cheaply and quickly than alternative slow backend databases [8] [15]. Therefore, a single caching server can replace tens of backend database servers by absorbing massive database request loads [8].

In-memory KV stores have three prominent use cases in data centers. 1) Storage caching 2) Computation caching 3) Transient data caching [15]. Storage caches are the most common use case of in-memory KV stores. These caches, which are typically read-heavy, substantially reduce latency and increase throughput [14] [6]. Furthermore, they alleviate the load of storage and databases systems. In-memory KV stores used for computation store past computation results (e.g., stream processing systems or ML predictions) for later uses [15]. These caches are typically write-heavy and ephemeral. In-memory KV stores used for transient data hold objects that merely live in the cache. Rate limiters are a good example of this type of cache [15].

Given certain hardware constraints, the value of in-memory KV stores is measured by their memory consumption, throughput, scalability, and power consumption [15]. Memory consumption determines the amount of memory a KV store needs to achieve a certain miss ratio. Throughput is measured in terms of the number of served requests per second. Scalability indicates how well a cache can use multiple cores on a host [15]. Power consumption determines the power needed by the KV store to achieve a specific throughput [4].

In this paper, we first highlights the characteristics of workloads of in-memory KV stores: The object popularity in most workloads follows Zipf distribution, indicating that KV store caches have a significant role in the quality of delivered services in these data centers [11] [14] [6]. The majority of object sizes in-memory KV stores are small. This might lead to a situation where objects' footprints are less than metadata footprints [12] [10]. Both Facebook [6] and Twitter [14] report that the workload of their in-memory KV stores follows a daily pattern, presenting an opportunity for elastic KV stores to rightsize the resources allocated to the KV store. Most workloads are read-heavy, indicating that concurrency can improve scalability and throughput substantially [11] [15]. TTLs are set for many objects hosted in-memory KV stores. Timely removals of expired objects enhance memory efficiency [15].

We discuss the shortcomings of current in-memory KV store designs adopted in today's data canters. Widely adopted in-memory KV stores such as Memcached and Redis suffer from internal and external memory fragmentation, respectively [8]. They impose a significant memory overhead (more than 50 B) per object to implement their policies [14]. Given that the vast majority of objects are small, the DRAM consumption of KV stores meta-

data exceeds the memory consumption of objects themselves. Current KV stores are not elastic, leading to low utilization and performance degradation in the presence of load variability [8] [4]. They hamper the scalability and throughput because of the extensive use of locking. They are not power-aware. Given that DRAM is energy-hungry, in-memory KV stores consume a large amount of power [4]. Finally, the tail latency of in-memory KV stores is of great concern as many services with high fan-out rely on them [5] [10].

We then present the state-of-the-art studies that attempt to address in-memory KV stores' shortcomings. Several works such Mica [11], Segcache [15], and Memshare [8] mitigate fragmentation by using log-structured memory management. Segcache [15] reduces object metadata by macro managing objects where a bulk of objects share metadata. Kangaroo [12] reduces the index's memory footprint by adopting a hierarchical design to achieve the best of both log-structured and set-associative designs. Memshare elastically allocates memory to enable consolidating multiple KV stores on the same machine to increase memory efficiency. Mica increases throughput and scalability by partitioning the key space, where each CPU has exclusive access to its partitions. Peafowl [4] reduces the power consumption of in-memory KV stores by transferring the CPU scheduler to the application to rightsize the number of CPU cores allocated to KV stores based on the load intensity, mitigating power consumption over low periods of utilization. Didona et al. [10] decreases the tail latency of in-memory KV stores by eliminating the head-of-line blocking, where large requests prevent small requests from being handled quickly.

Although the discussed research works significantly improve performance, scalability, memory, and power efficiency, they are far from ideal. Today's date center application requirements (e.g., the need for non-flat data structures), workload characteristics (e.g., the existence of write-heavy workloads), and novel hardware (e.g., persistent memory) call for the design of novel KV stores more tailored for today's data center requirements. We conclude our survey with future research directions (open problems).

2 Background

As shown in Figure 1, an in-memory KV store such as Memcached consists of four main components [2]:

1. A dispatcher thread that gets requests from user APIs and distributes the requests among worker threads in a round-robin fashion.

2. The worker threads serve the user requests (e.g., get, put, delete) using an event-driven architecture

3. The hash-table is simply an array of buckets where each bucket holds a pointer to a chain of items hashed to that bucket.

4. The memory component that manages the memory.

Memcached uses slab classes to allocate memory. Each slab is simply a big chunk of memory the is divided into smaller chunks of fixed size. Each class is responsible for a specific size. For example, slabs in the first-class host chunks of size 96 bytes and slabs in the second class are responsible for chunks of size 120 bytes. Chunks in each slab are organized as a linked list (i.e., LRU list), where the head of the linked list is an item that is most recently used, and the item at the tail of the queue is an item that is least recently used. Every read and write rearranges the LRU, moving the recently used item to the head of the list. On eviction, items located at the tail of the LRU list have the highest priority to be evicted from the memory.

2.1 Memcached operations

Figure 1 shows the path of read and write operations in Memcached. On a read operation, the user request is assigned to a worker (1)(2). The worker finds the hash bucket of the request's key (3). The worker then traverses the bucket's chain (4) to find the key. If found, the worker retrieves the item pointed by the corresponding hash bucket element and returns it to the user. The read operation will finally update the LRU queue.

On write operations, the user request is first assigned to a worker by the dispatcher thread (a) (b). The worker finds the hash bucket of the new item (c). The key is linked to the hash bucket's chain (e) if the key does not exist. The worker then finds the class of item based on its size (d). If the class has a slab with an empty chunk, the item will be placed there, and the hash table is updated with the address of the item. Otherwise, a new slab must be allocated. If there exists no new slab, eviction is triggered. Finally, the LRU queue of the slab must be updated to move the newly added item to the head of the LRU queue.

Besides set and get (the most common operations), Memcached also supports Replace, Append, Cas, Delete, Incr, and Decr. Get takes a key and retrieves its corresponding item. Set stores the new item, possibly overwriting any existing data (the new item is added to the head of the LRU list). Appends adds new data at the last byte of an existing item. Cas (Check And Set) stores data only if no one else has updated the data since the user read it last. Deletes remove the item from the cache if it exists. Incr and Decr increase and decrease the item's value, respectively (item must be an integer) [2].

2.2 Redis

Redis [3] is a single-threaded KV store. When used as a cache, similar to Memcached, it leverages a hast table for O(1) look ups. Redis uses external memory allocators such as Malloc for memory allocation. On eviction, it randomly chooses a limited number of keys (e.g., five keys). Redis compares these keys to a pool of best candidates (typically 16 candidates) for eviction based on their idle time. These candidates have the highest idle times. Finally, a candidate item with the highest idle item is chosen to be evicted.

3 Workload Characterization

In this section, we describe the main characteristics of workloads of in-memory KV stores and their implication on the design of KV stores.

3.1 Object popularity

Object popularity is an essential workload characteristic of in-memory KV store caches. For example, if the workload follows Zipf distribution, the cacheability of the workload is much higher than the workload with uniform distribution. Both Twitter [14] and Facebook [6] report that the majority of workloads in Twitter and Facebook follow Zipf distribution (i.e., informally, 80% of requests is for 20% of objects), meaning that Zipf distribution frequency-rank cure in log-log scale is linear. This indicates caches have a significant role in the quality of delivered services in these data centers.

3.2 Churn

Churn refers to the change in the working set due to the introduction of new keys and the popularity changes of existing keys over time. Facebook reports that there exists a high degree of churn across all workloads in Facebook [7]. Over two-thirds of popular objects in a given hour fall out of the top 10% after just one hour. High churn reduces the effectiveness of caching mechanisms that estimate object popularity based on past access patterns.

3.3 Object Size

Several studies have reported that the majority of object sizes in in-memory KV stores are small [14] [6]. Twitter [14] says that 85% of key sizes are smaller than 50 bytes. They also report that value sizes are variable and range from 10 bytes to 10KB.

Facebook also reports that in their SocialGraph caches, a significant fraction of objects is between 10B and 20B, while big objects exist (e.g., 64KB and 128KB objects) [7]. Object sizes play a crucial role in the performance of in-memory KV stores. For instance, as the size of objects becomes smaller, the size of the index and caching metadata becomes bigger, possibly occupying more space than objects themselves [12].

3.4 Burstiness and Diurnal pattern

Both Facebook and Twitter report that the workload of their in-memory KV stores follows a daily pattern [14] [6]. They also note that the requests arrival rate varies much more than Poisson suggests (default assumption in system evaluations). Additionally, Facebook's traffic is quite bursty. Caches in Facebook, for instance, have sharp bursts. Bursty arrival rates make it challenging to provide caching systems with sufficient resources to maintain expected performance during load spikes. On the other hand, the daily workload pattern allows elastic KV stores (if realized) to rightsize the resource and power consumption based on the offered load [4].

3.5 Workload Composition

Twitter reports that get and set are the most dominant operations in-memory KV stores [14]. They note that 90% of the operations are read operations, indicating that most caches serve read-heavy workloads. However, a notable portion of operations is write operations. Twitter reports that more than 35% of Twitter cache nodes are write-heavy[1]. The workload composition is an essential factor in designing KV stores. For example, enhancing scalability by increasing thread concurrency for caches used for the write-heavy workloads is challenging due to the overhead of thread synchronization [15] [11].

3.6 TTL

Memcached and Redis widely use Time-to-live (TTL) to meet data freshness or comply with regulations such as GDPR [15]. TTL distinguishes in-memory key-value stores from persistent key-value stores. TTL is set when an object is created. It indicates the expiration time of the newly created objects. Request attempt to access an expired object is a cache miss. Hence, keeping expired objects in the cache is not valuable. In Twitter, TTLs range from a few minutes to a month. They report that 25% of the objects use TTL of less than twenty minutes, and 25 % of the objects have TTL of higher than two days. Timely removal of expired objects is of great importance as it leads to higher memory efficiency and hit rate [14] [15].

[1] Twitter defines a cache node write-heavy if the percentage sum of the set, add, cas, replace , append , pre-append, incr and decr exceeds 35%

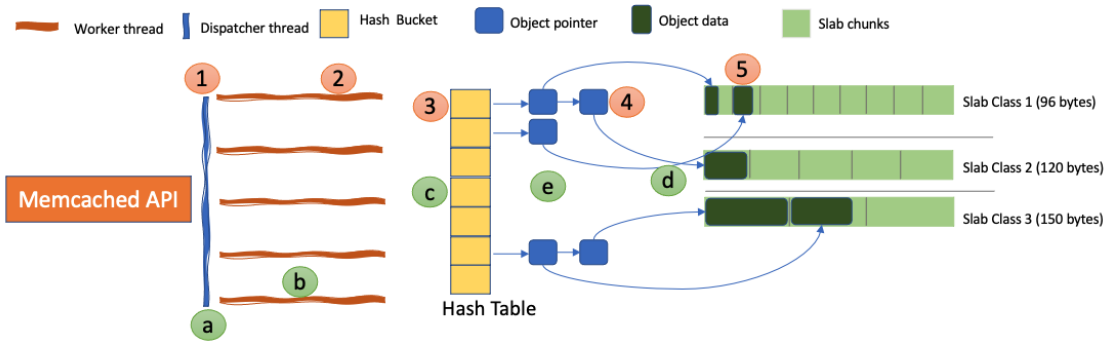

Figure 1: The Architecture of Memcached

Summary: The object popularity in most workloads follows Zipf distribution. The majority of object sizes are small. The workload intensity follows a daily pattern. Most workloads are read-heavy. TTLs are set for most objects hosted in in-memory KV stores.

4 Memory Management

Memory management is one of the fundamental design aspects of in-memory KV stores. DRAM is expensive and energy-hungry. Therefore, the efficient use of DRAM is a matter of great concern [8] [12] [15]. Many studies increase memory efficiency by improving the 1) admission and 2) eviction algorithms, 3) using novel memory allocators [11] 4) reducing the object metadata [12], and 5) removing expired objects in a timely manner [15]. In this paper, we focus on the last three techniques.

4.1 Memory Allocation Policies

In-memory KV stores mainly use three policies for memory allocation: 1) External memory allocation, 2 Slab-based memory allocation, or 3) log-structured memory allocation [8]. Table 1 summarizes the pros and cons of each approach.

4.1.1 External Memory Allocators

This approach allocates the memory from the heap on demand (e.g., using malloc). This approach is straightforward. However, on-demand heap memory allocation leads to external fragmentation [11]. This is because several research works have reported that the workload of in-memory KV stores often consists of small and variable-sized objects [14] [12]. Therefore, systems such as Redis with external memory allocators are vulnerable to external memory fragmentation and OOM (Out of memory) [14].

4.1.2 Slab-Based Memory Allocator

This approach allocates memory in fixed-sized chunks known as slabs. As mentioned before, Memcached leverages this approach. It uses the default size of 1 MB for its slabs. Each slab is then evenly divided into smaller chunks of memory called items. The class of each slab determines the size of its items. Lower slab classes hold small items (the default minimum is 88 bytes), and higher ones keep more oversized items. An object is mapped to a slab class that best fits it.

Slab-based memory allocation does not have external fragmentation; however, it can lead to internal memory fragmentation at the end of each chunk and the end of each slab. Defragmentation can pack objects to make free memory spaces, but it involves expensive memory copies. In addition, slab-based allocators introduce the slab calcification problem, where a slab class cannot get enough memory and exhibit higher miss ratios. This is because slabs are assigned to classes in a first-come-first-serve manner. Therefore, the newly popular slab classes cannot find more memory when the popularity of slabs changes because all slabs have already been assigned. To solve this problem, Memcached migrates slabs between classes; however, it is not always effective. Migrating slabs may increase the miss ratio because all objects on the outgoing slab are evicted [15].

4.1.3 Log-Structured Memory Allocator

Log-structured memory allocators place new data items at the end of a linear data structure called a log. To update an item, the new item is inserted to the tail of the log, which overrides the previous value. Hence, inserts and updates access memory sequentially, incurring fewer cache misses, making logs write-friendly particularly for bulk data writes. More importantly, this approach leads to lower memory fragmentation [11][15]. Nevertheless, in log-structured designs, space occupied by overwritten and deleted objects must be periodically reclaimed. A garbage

Table 1: Comparison of memory allocation policies

Memory Allocator	External Fragmentation	Internal Fragmentation
Malloc-Based	Yes	No
Slab-Based	No	Yes
Log-structured	No	No

collector moves live objects to a new log and removes the old log. Garbage collection is costly and often reduces performance because of the large amount of data it must copy [11].

Mica [11] is an in-memory KV store that employs log-structured memory allocation to mitigate the memory fragmentation caused by malloc- and slab-based systems. Mica employs a circular log to reduce the cost of garbage collectors. Mica's design reduces memory fragmentation; however, Mica is limited to using basic eviction algorithms (e.g., FIFO), which degrades the hit ratio [15].

Summary: Malloc- and slab-based memory allocators leveraged by widely used in-memory KV stores such as Memcached and Redis lead to external and internal memory fragmentation. Log-structures designs eliminate memory fragmentation at the cost of collecting garbage (dead objects) periodically.

4.2 Object Metadata

Both Twitter [14] and Facebook [6] report that the majority of objects stored in-memory KV stores are small. For instance, Twitter's top four production clusters' mean object sizes (key+value) are 230,55, 294, and 72 bytes, respectively [14]. Current popular production caching systems store a relatively large amount of metadata per object to realize their caching policies [12]. For example, both Memcached and Redis impose over 50 bytes of memory overhead per object [15]. These metadata are critical for caches' mechanisms and cannot be dropped without removing some functionalities or features. Moreover, research aimed at reducing the miss ratio using sophisticated policies typically expands object metadata even further [15].

Segcache [15] attempts to solve this problem by leveraging several techniques. It uses a log-structured design where the log is sliced into smaller chunks of memory, called segments. Each segment (a bulk of objects) shares metadata, such as creation time and TTL. In addition, it uses bulk changing (described in section 5) where per-object pointers needed for hash table collision chains are eliminated. Finally, it macro mange the cache (segment granularity) to eliminate the need for per-object concurrency metadata. Combining these techniques, Segcache significantly reduces object metadata.

4.3 Expired Objects

Expired objects are not valuable, quick removal of expired objects improves memory efficiency and hit ratio [15]. In-memory KV stores mainly take the following approaches for expired object removal (summarized in Table 2).

Lazy expiration: This approach removes objects on access. If an expired object is accessed, it is deleted from DRAM. Lazy expiration is simple. However, it reduces memory efficiency because it does not remove expired objects promptly.

LRU queue This approach checks a fixed number of objects at the tail of the LRU queue to remove the expired ones. This approach has several flaws: 1) Using LRU queue hampers thread scalability due to the extensive use of locking for concurrent accesses to LRU queue [11]. 2) This approach is opportunistic and therefore doesn't guarantee the timely removal of expired objects [15]. 3) Many production caches track billions of objects. Therefore, the time for an object to get to the tail of the LRU queue can be very long, delaying the timely removal of expired objects [15].

Full cache scan is another approach adopted by Memcached. This solution periodically scans all the cached objects to remove expired ones. Frequent full cache scans can guarantee the timely removal of expired objects. However, this approach is costly and significantly wastes resources [15].

Random sampling is adopted by Redis. This approach periodically samples a subset of objects and remove expired ones. This approach is expected to be less costly compared to a full scan. Surprisingly, the cost can be higher than a full cache scan due to excessive random memory access. Moreover, this approach is still opportunistic as it can not ensure the timely removal of all expired objects.

Segcache [15] is a state-of-the-art solution whose primary goal is the timely removal of expired objects with minimum overhead. To this end, Segcache first groups objects into segments (1 MB default size). A segment in Segcache is similar to a small log in log-structured systems. All objects located in a segment have approximately the same TTL.

On the other hand, Segcache breaks the spectrum of possible TTLs into ranges. Seagate then uses an array to index segments based on approximate TTL. Each element in this array is called a TTL bucket. Each non-empty TTL bucket points to the head and tail of a time-sorted segment chain, with the head segment being the oldest.

Table 2: Comparison of approaches designed for removing expired objects

Expired objects removal	Lazy Expiration	LRU queue	Full Scan	Random sampling
Memory efficiency	Low	Medium	High	Medium
Scalability	High	Low	Low	Medium
Overhead	Low	Medium	High	Medium

Segcache uses a background thread to scan the TTL buckets in order. If the first segment of a TTL bucket is expired, the background thread removes all the objects in the segment (macro managing). The background thread continues down the chain until it runs into one segment that is not yet expired. Segcache's proactive expiration technique promptly recycles the memory occupied by expired objects, improving memory efficiency.

Summary Since most object sizes are small, reducing per-object metadata can significantly mitigate DRAM consumption. Additionally, given that a vast majority of objects are associated with TTLs, timely removal of expired objects can further enhance memory efficiency.

5 Elastic Memory Allocation

Allocating an entire machine to a KV store to serve an application hampers utilization. Multitenant KV store servers are an excellent solution to enhance data center utilization, where multiple KV stores that serve different applications are all consolidated in one server [8].

Caching-as-a-service providers like Memcachier allow customers purchase a fixed amount of memory. Each application is statically allocated memory, and a separate eviction queue for each application is maintained. Static memory partitioning is straightforward; However, it degrades the memory efficiency and performance. Tenants that do not fully utilize their share wastes memory and tenants that temporarily need more than fair share (e.g., due to bursts) experience performance degradation [8].

An elastic KV store that provides performance isolation is an ideal solution. This elastic KV store should allocate unused memory resources when the application has a burst of requests and take back memory when it no longer needs it.

Memshare [8] is an elastic KV store designed to address the inefficiency of static partitioning approaches. Memshare consists of two main components. Memshare's judge determines how much memory should be assigned to each application. The second component is Memshare's cleaner, which dynamically allocates memory by prioritizing eviction from applications that use memory more than they need.

Memshare cleaner is a background thread that chooses applications with high memory consumption and evicts their least useful items using LRU queues. Memshare pools free memories to dynamically allocate them to applications with higher needs.

Memshare estimates the hit rate curve for each application to find the fair share of memory for each application dynamically. This curve indicates the hit rate the application would achieve for a given amount of memory. Knowing applications' hit rate curves, Memshare allocates more memory to applications whose hit rate would benefit the most.

To estimate the hit rate curve, Memshare uses shadow queues. A shadow queue of an application holds a subset of object keys of the applications. Object evicted for an application cache is inserted to shadow queue. The shadow queue hit rate approximates an application's hit rate curve. The application with the highest hit rate in its shadow queue would benefit from a higher memory share.

Summary Elastic memory allocation allows for multi tenancy and, therefore, higher utilization. However, designing an arbiter that dynamically quantifies the fair memory share and quickly allocates the memory while ensuring isolation is challenging.

6 Indexing

Common choices for indexing in in-memory KV stores are hash tables and tree-like structures. These data structures deliver better performance to reads compared to writes. Hash tables examine several slots to find space for the new item, and trees may require multiple operations to maintain structural invariants [11].

Hash tables that use chaining are more writer-friendly because they insert new items at the tail of the chain without accessing many memory locations. However, they degrade the lookup performance because it traverses a long chain of items, leading to multiple random memory accesses.

Besides the read and write throughput, the index size is crucial in designing an in-memory KV store. When the

object sizes are tiny, the size of the index may take most of DRAM [12].

Random-access memory of traditional hash tables hampers scalability and throughput [11]. Collision resolution requires walking down the hash chain, leading to multiple random DRAM accesses and string comparisons. Further, chaining imposes a memory overhead of an 8-byte hash pointer per object, which is notable compared to small object sizes [12].

Mica [11] and Segcache [15] solve this problem using bulk-chaining hash tables. They allocate eight slots (64 B - a cache line) to each bucket. The first slot stores the bucket information, the following six slots store object information. The last slot is either abject information or a pointer to the next hash bucket. Bulk chaining decreases object metadata by removing hash pointers and improves lookup throughput by reducing random DRAM accesses.

Summary Traditional (chain-based) hash tables employed by Memcached and Redis reduce throughput while increasing memory overhead. On the other hand, bulk-chaining hash tables decrease metadata and improve throughput.

7 Scalability and Throughput

In-memory KV stores can leverage modern multi-core systems to increase scalability and throughput [15]. Faster servers reduce the number of needed machines. Consequently, they mitigate the cost, power consumption, and data center footprint. However, current KV stores do not scale well. Memcached, for instance, cannot scale as expected because of extensive locking used for LRU queues, free object queues, and the hash table.

Current in-memory KV stores broadly adopt two main techniques to improve scalability: 1) Parallel Data Access and 2) Exclusive access.

Parallel Data Access. Multiple CPU cores can access the shared data. The integrity of the data structure is maintained using mutexes, optimistic locking, or lock-free data structures. This approach increases throughput for both reads and writes operations. However, concurrent writes scale poorly due to locking overhead and frequent cache line transfer between cores. Only one core can hold the cache line of the same memory location for write operations [11].

Exclusive Data Access. This approach shards the data. Each core exclusively accesses its own partition in parallel without inter-core communication. Partitioning can lead to good throughput and scalability; however, it degrades performance when the load between partitions is imbalanced (e.g., when the key popularity is skewed). Additionally, since each core can access only data within its partition, request direction might be needed to forward requests to the appropriate CPU core, which is costly [11].

Current KV stores mitigate the lock overhead and use opportunistic concurrency controls and epoch-based systems to increase the scalability in the parallel data access approach. They use data partitioning and DRAM partitioning to improve scalability in the exclusive access approach. All of these approaches improve scalability to some extent; however, they all come with costs: Reducing the locking overhead might be realized by ignoring functionalities. For example, Memcached can have a simpler eviction policy by replacing the LRU queue with FIFO queues, improving scalability at the cost of lower memory efficiency. Data partitioning may lead to load imbalance and less efficient resource utilization. Static DRAM partitioning uses memory inefficiently. Opportunistic concurrency control with lock-free data structures does not work well with write-heavy workloads. Epoch-based systems require a log-structured design with a sub-optimal eviction algorithm [15].

Segcache [15] improves scalability by using a combination of techniques such as minimal critical sections and optimistic concurrency control. To this end, it replaces the object-level bookkeeping of current KV stores with segment-level bookkeeping. In other words, only segment chain modification needs locking, notably reducing the critical sections.

Mica[11] increases scalability by partitioning the data, where each CPU has exclusive access to its partitions. Mica exploits CPU caches and packet burst I/O to speed up CPU's responsible for more loaded partitions, reducing the impact of skewed workloads. More specifically, Mica claims that the impact of load imbalance is not notable. Because hot partitions creates locality in the data path, experiencing fewer CPU cache misses when accessing items.

Mica is able to fall back to concurrent reads if the load is highly uneven. However, it avoids concurrent writes, which are always slower than exclusive writes.

Mica improves throughput by eliminating synchronization and inter-core communication, making Mica scale linearly with CPU cores; however, Mica relies on Flow Director to partition KV store requests among cores. Therefore, Mica's clients have to encode object-level affinity (e.g., objects keys) information in a way Flow Director can understand.

Summary Scalability can be notably enhanced by leveraging parallel data access or exclusive data techniques. However, parallel data access scales poorly for write-intensive workloads due to locking overhead, and exclusive data access techniques do not scale well for skewed workloads.

8 Power-proportional KV stores

Facebook's ETC workload follows a diurnal pattern with 2× load variations over a day [6]. Twitter also reports the same behavior [14]. Long-term variations present an opportunity to leverage power-proportional KV stores to rightsize resources (e.g., CPU) allocated to KV stores to mitigate power consumption.

Energy-proportional KV stores must have three essential properties: Energy proportionality, Microsecond-scale tail latency, and ease of deployment and generality [4].

Energy Proportionality. An energy-proportional KV store exploits load variability to enforce energy proportionality by scaling the processing capacity to match the offered work, considerably reducing power consumption during low utilization periods. Given the widespread adoption of in-memory KV stores, low power consumption of cache nodes will reduce costs, data-center footprints, and environmental impacts.

Microsecond-scale tail latency: Many applications with a high fan-out pattern rely on in-memory KV stores. Therefore, the quality of delivered services is determined by tail latency [5] [9]. Hence, an energy-proportional KV store must keep the tail latency of KV stores at a microsecond scale.

Ease of deployment and Generality Solutions that rely on significant modifications to the OS or device drivers are less practical as they transform general-purpose machines to single-purpose ones. An elastic solution must be readily deployable in data centers [4].

8.1 Existing Approaches

Broadly speaking, existing approaches for power-proportional KV stores can be classified into the following categories (summarized in Table 3):

Idle states. CPUs feature several power-saving modes called c-states (idle-states) to save power during idle periods (e.g., C0 (shallowest), C1, C2, C6 (deepest)). C0 is the operational mode when the CPU executes instructions and saves no power. C6 is the deepest cState which saves the most power; however, the CPU will require more time to wake up from this state (133s Xeon 6130). Due to the latency impact of deep idle states, CPUs enter a particular c-state only when it predicts the next idle period is greater than a threshold, known as the target residency (600 us for C6 is Intel Xeon 6130) [13]. cStates can notably save power for applications with long idle periods. However, in-memory KV stores typically receive high arrival rates that fragment idle periods into very short idle cycles that deep states (e.g., C6) cannot exploit, making cStates notably less effective for power saving [4].

Feedback-based controllers monitor and measure the offered load to the KV store, compare it to predefined thresholds and scale the number of allocated cores to save power during low utilization periods. Feedback-based controllers can adapt to diurnal variations; however, they are too slow (operating at second-scale intervals) to cope with short-term burstiness, lengthening tail latency.

DVFS-based approaches exploit the latency slack (i.e., SLA latency minus current latency) so as to slow down processing using DVFS to save power while executing requests in time. This approach saves power for applications with large service time; however, the short service times (less than ten us) and high arrival rates of KV stores do not provide much opportunity for power saving with this approach.

8.2 Peafowl

Peafowl [4] transfers the scheduler from the operating system to the application (i.e., KV store), where there is more domain-specific knowledge and control. Peafowl leverages a monitoring system to identify off-peak periods. Knowing off-peak periods, its scheduling policy consists of two main parts: scale-down and scale-up. Peafowl's scale-down process consolidates connections onto fewer CPU cores, allowing inactive cores to exploit deep idle (e.g., C6) state to save power during low or medium utilization periods. Peafowl's scale-up process expands the allocated cores to avoid increasing tail latency when the load approaches its peak.

Peafowl Scale-down process. In Peafowl, each worker (KV store thread) periodically reports their current load to the scheduler. When the scheduler sees that the total load can be packed into fewer workers (i.e., cores), it starts the scale-down process. Peafowl uses a greedy algorithm for the scale-down process. It chooses the worker with the lowest load as the scale-down worker. It then instructs the scale-down worker to give up its connection to other workers gradually. Once all the scale-down worker's connections are migrated, Peafowl enables the idle-states on the scale-down worker to save power.

Peafowl Scale-up process. When a worker's load approaches its load limit (learned by Peafowl's monitoring system), they start transferring connections back to another worker. In Peafowl, workers do not coordinate with the scheduler for performing the scale-up migration. This

Table 3: Comparison of power saving approaches

Approach	Power saving	Tail latency
Idle states	low	low
DVFS and Request delaying	low	medium
Feedback-based controllers	high	high

leads to immediate scale-up, thereby avoiding tail performance degradation.

Summary The high arrival rate and short service time of in-memory KV stores do not let traditional power-saving approaches like idle states or DVFS save power. These approaches might also length the microsecond level tail latency of in-memory KV stores.

9 Hybrid KV Stores

In today's data centers, small objects are prevalent. The average object size is less than 700B. At Twitter, for example, the average tweet size is less than 33 characters [14] [12]. While individual objects are tiny, application working sets still add up to TBs of data, which needs a significant amount of DRAM. DRAM is, however, expensive and power-hungry. Therefore, the trend in data centers is towards less DRAM and more Flash (or other medias) use [12].

Flash is persistent, cheaper, and more power-efficient than DRAM. Flash, however, has limited write endurance, which means there is a limit on the number of writes before the Flash wears out. Besides, Flash suffers from write amplification [1]. Write amplification occurs when the number of bytes written to the underlying Flash exceeds the original number of bytes. Finally, Flash can be read and written only at multi-KB granularity (e.g., 4KB). For example, writing a 100 B object requires writing a 4 KB flash page, amplifying bytes written by 40× [12]. There are two common approaches (Summarized in Table 4) for caching billions of objects on Flash: log-structured designs with index in DRAM and set-associative designs.

9.1 Log-Structured Designs

One approach to deal with billions of tiny objects is to employ log structure designs with an index in DRAM to track objects. This approach reduces write amplification and works well for larger objects; However, it requires large amounts of DRAM when objects are small, as the index must keep one entry per object. Therefore, the total DRAM required for a log-structured are high and therefore increase cost and power.

9.2 Set-Associative Designs

Another approach is to use set-associative caches with a small DRAM footprint. In this approach, a hash function maps each object to a specific 4 KB set (i.e., a flash page). Therefore, it reduces the metadata needed to locate objects on Flash by restricting their possible locations. For example, Facebook's CacheLib [7] is a set-associative cache that keeps only three bits per object (used for bloom filters) in the DRAM.

Set-associative designs reduce DRAM consumption. However, they significantly increase the write amplification. Inserting a new object into a set means rewriting an entire flash page, most of which is unchanged. For example, writing 100B objects requires writing 4 KB page, 40x higher write amplification, notably reducing device lifetime [12].

9.3 Kangaroo

Kangaroo [12] is a flash-based cache designed for billions of tiny objects at Facebook. Kangaroo adopts a hierarchical design to achieve the best of both log-structured and set-associative designs to minimize DRAM usage and Flash writes. Kangaroo consists of two main components: Klog (a log-structured flash cache) and Kset (a set-associative flash cache)

Klog is a log-structured design whose goal is to reduce write amplification. It writes objects in a circular buffer on Flash in batch and tracks them using a small index located in DRAM. Every once in a while, a portion of kLog (i.e., segment) is dumped to kSet. Klog treats all objects mapped to the same bucket in its index as an entry for kSet. By batching all elements that can be mapped to a position in kSet (a Flash page), Klog notably reduces write amplification.

KSet's role is to minimize the DRAM footprint of the cache. KSet employs a set-associative cache design where the cache is sliced into sets of 4KB. kSet map an object to set by hashing its key, eliminating the need for DRAM indexes. In Kangaroo, kSet also keeps a small bloom filter in DRAM to reduce unnecessary flash reads.

Summary When it comes to caching of billions of tiny objects on flash, log-structured design needs significant DRAM, although they reduce write amplification, and set-associative techniques increase write amplification while they reduce DRAM footprint.

Table 4: Comparison of Hybrid cache approaches

Approach	DRAM Consumption	Write Amplification
Flash-Log structured	High	Low
Flash-Set-associative	Low	High

10 Tail Latency

Many distributed applications that use in-memory KV stores exhibit a high fan-out pattern, i.e., they issue a large number of requests in parallel. From the application's standpoint, the overall response time is determined by the slowest responses to these requests, hence the tail latency of in-memory KV store is critical [5] [8] .

Tail latency has many sources. One source of latency is head-of-line blocking [10] [5]. As mentioned before, in-memory KV store workloads consist of both small and oversized items. Head of the line blocking is a situation in which a request for a small item ends up waiting while a large item is being processed, which may increase the tail latency [10].

Didona et al. [10] introduces the notion of size-aware sharding to address the head of the line blocking. They assigned objects to disjoint sets of cores, where some cores serve small objects, and some cores are responsible for serving large objects. By isolating the requests for small items, they do not experience any head-of-line blocking. Given the fact small objects are account for a substantial percentage of requests, the corresponding percentile of the latency distribution is improved.

To this end, Didona et al [10] classifies core into large and small cores. Cores handling small requests are called small cores, and core handling large requests are called large cores. To process a request, small cores examine the size of the object of the requests. The request is handled in the small core if its size is smaller than a threshold. Otherwise, the request is inserted into the request queue for large cores. On the other hand, large cores fetch requests from the request queue of large cores and process them one by one. By doing so, they isolate small objects from large objects, eliminating the head of the line blocking and improving tail latency. In addition, this approach dynamically monitors the request size distribution to assign appropriate number core to small and large objects.

Summary Many services with high fan-out patterns rely on in-memory KV stores. Hence, the tail of the latency distribution in in-memory KV stores is a matter of great concern.

11 Future research

11.1 Elasticity

As Twitter and Facebook, note the workloads of their in-memory KV stores follow a daily pattern, and working set size changes over time [6] [14]. Therefore, elastic KV stores that can rightsize both CPU and memory consumption of KV stores can notably reduce the cost, power consumption, and environmental impacts of KV store clusters. However, realizing an elastic KV store in the presence of bursts and micro-second level SLAs is challenging. Therefore, many data centers provide more capacity than needed to ensure high quality of user facing service, notably wasting resources. Peafowl [4] is an attempt toward elastic in-memory KV stores. It elastically assigns CPUs based on the intensity of offered load; however, it is agnostic of working set sizes and memory consumption. This calls for more research toward a true elastic KV store that rightsizes both memory and CPU while not violating SLA.

11.2 Persistent memory

Hybrid KV stores offer significantly higher capacity than DRAM, where a large portion of the working set size resides on another medias (e.g., Flash). As a consequence, the hit ratio is drastically increased. However, media such as Flash has lower read and write throughput, and random writes decreases their lifespan. Kangaroo [12] is an attempt to build KV stores on Flash by reducing the number of random writes and write amplification. On the other hand, persistent memory (pMem) has its own unique characters. Persistent memory has significantly lower write amplification compared to Flash. They allow for a much larger number of lifetime write cycles and thus a higher endurance than Flash. They are byte-addressable like DRAM, non-volatile like Flash and have a read/write latency between the two. The reads throughput is only two times less than DRAM, and the random write throughput is ten times less than memory [1]. Therefore, designing pMem-based KV stores deserves more explorations and research.

11.3 Write-heavy workloads

As twitter reports, a significant portion of their KV stores is write-heavy (35% of the 153 cache clusters) [14]. However, most existing systems, designs, and research work assume a read-heavy workload. Write path is usually costly (e.g., because of locking overhead) and can trigger more expensive events such as eviction. Twitter reports that serving write-heavy workloads has higher tail latencies than ready-heavy ones. In addition, the scalability in

write-heavy caches is challenging to achieve. Mica [11], for example, distributes the workloads among cores based on their key hashes. Each core is exclusively responsible for writing a portion of the key space. Mica delivers good performance for read-heavy workloads; however, they do not explore the scalability of Mica for write-heavy workloads. This calls for future research on designing systems tailored for write-heavy workloads.

11.4 Non-flat data structures

Most of today's in-memory KV stores are designed for flat data structures. On the other hand, today's application in data centers needs complicated data structures such as array, list, dictionary, and heap. Time series applications, for example, keep a list of entries for each key in the memory. Redis supports time series, but it does offer any memory management strategy tailored for this data structure. We believe that KV stores must be tailored for different data structures based on their characteristics to achieve better memory efficiency and higher performance.

12 Conclusion

Table 5 summarizes the research works investigated in this paper. It broadly classifies the challenges of in-memory KV stores in current data centers into memory efficiency, scalability, throughput, elasticity, supporting persistent media, offering non-flat data structures, and supporting write-heavy workloads, and compares surveyed works to Memcached. From the table, existing approaches are far from ideal. Each tackles a subset of problems and improves them to some extend. This calls for research on designing more tailored in-memory KV stores for current data centers. However, given the complexity of application needs (e.g., non-flat data structures) and offered workloads (e.g., write-heavy workloads), the tight SLA requirements (e.g., micro-second level tail latency), different hardware characteristics (e.g., write amplification of Flash devices), and the existence of bursty workloads realizing an ideal KV store for data centers is challenging.

References

[1] Intel® optane™ technology delivers new levels of endurance. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/delivering-new-levels-of-endurance-article-brief.html. Accessed: December 2021.

[2] Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: December 2021.

[3] Redis. https://redis.io/. Accessed: December 2021.

[4] E. Asyabi, A. Bestavros, E. Sharafzadeh, and T. Zhu. Peafowl: In-application cpu scheduling to reduce power consumption of in-memory key-value stores. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 150–164, New York, NY, USA, 2020. Association for Computing Machinery.

[5] E. Asyabi, S. SanaeeKohroudi, M. Sharifi, and A. Bestavros. Terriertail: Mitigating tail latency of cloud virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2346–2359, 2018.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[7] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, Nov. 2020.

[8] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[9] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, feb 2013.

[10] D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, Feb. 2019. USENIX Association.

[11] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*

Table 5: A classification of surveyed works compared to Memcached

	Mica	Kangaroo	Segcache	Sharding [2]	Memshare	Peafowl
Memory Efficiency	Equal	Lower	Lower	Equal	Equal	Equal
Scalability	Higher	Higher	Higher	Equal	Equal	Equal
Throughput	Higher	-	Higher	Higher	Equal	Equal
Latency	Lower	-	-	lower	-	equal
Elastic DRAM	No support	No support	No support	No support	Support	No support
Elastic CPU	No support	No support	No support	No support	No Support	Support
Flash support	No support	Support	No support	No support	No Support	No Support
pMem support	No support	No support	No support	No support	No Support	No Support
Write-Heavy Workloads	No support	No support	No support	No support	No Support	No Support
Non-flat data structures	No support	No support	No support	No support	No Support	No Support

(NSDI 14), pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[12] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[13] E. Sharafzadeh, S. A. S. Kohroudi, E. Asyabi, and M. Sharifi. Yawn: A cpu idle-state governor for datacenter applications. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 91–98, New York, NY, USA, 2019. Association for Computing Machinery.

[14] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, Nov. 2020.

[15] J. Yang, Y. Yue, and R. Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, Apr. 2021.