BlueCache: A Scalable Distributed Flash-based Key-value Store

Shuotao Xu[‡] Sungjin Lee[†] Sang-Woo Jun[‡] Ming Liu[‡] Jamey Hicks[♭] Arvind[‡]

[‡]Massachusetts Institute of Technology
{shuotao, wjun, ml, arvind}@csail.mit.edu

[†]Inha University
sungjin.lee@inha.ac.kr

[♭]Accelerated Tech, Inc
jamey.hicks@accelerated.tech

ABSTRACT

A key-value store (KVS), such as memcached and Redis, is widely used as a caching layer to augment the slower persistent backend storage in data centers. DRAM-based KVS provides fast key-value access, but its scalability is limited by the cost, power and space needed by the machine cluster to support a large amount of DRAM. This paper offers a 10X to 100X cheaper solution based on flash storage and hardware accelerators. In BlueCache key-value pairs are stored in flash storage and all KVS operations, including the flash controller are directly implemented in hardware. Furthermore, BlueCache includes a fast interconnect between flash controllers to provide a scalable solution. We show that BlueCache has 4.18X higher throughput and consumes 25X less power than a flash-backed KVS software implementation on x86 servers. We further show that BlueCache can outperform DRAM-based KVS when the latter has more than 7.4% misses for a read-intensive application. BlueCache is an attractive solution for both rack-level appliances and data-center-scale key-value cache.

1. INTRODUCTION

Big-data applications such as eCommerce, interactive social networking, and on-line searching, process large amounts of data to provide valuable information for end users in real-time. For example, in 2014, Google received over 4 million search queries per minute, and processed about 20 petabytes of information per day [26]. For many web applications, persistent data is kept in ten thousand to hundred thousand rotating disks or SSDs and is managed using software such as MySQL, HDFS. Such systems have to be augmented with a middle layer of fast cache in the form of distributed in-memory KVS to keep up with the rate of incoming user requests.

An application server transforms a user read-request into hundreds of KVS requests, where the *key* in a key-value pair represents the query for the backend and the *value* represents the corresponding query result. Facebook's memcached [52] and open-source Redis [5], are good examples of such an architecture (Figure 1). Facebook's memcached KVS cluster caches trillions of objects and processes billions of requests per second to provide high-quality social networking services for over a billion users around the globe [52].

Proceedings of the VLDB Endowment, Vol. 10, No. 4

KVS caches are used extensively in web infrastructures because of their simplicity and effectiveness.

Applications that use KVS caches in data centers are the ones that have guaranteed high hit-rates. Nevertheless, different applications have very different characteristics in terms of the size of queries, the size of replies and the request rate. KVS servers are further subdivided into application pools, each representing a separate application domain, to deal with these differences. Each application pool has its own prefix of the keys and typically it does not share its key-value pairs with other applications. Application mixes also change constantly, therefore it is important for applications to share KVS for efficient resource usage. The effective size of a KVS cache is determined by the number of application pools that share the KVS cluster and the sizes of each application's working set of queries. In particular, the effective size of a KVS has little to do with of the size of the backend storage holding the data. Given the growth of web services, the scalability of KVS caches, in addition to the throughput and latency, is of great importance in KVS design.

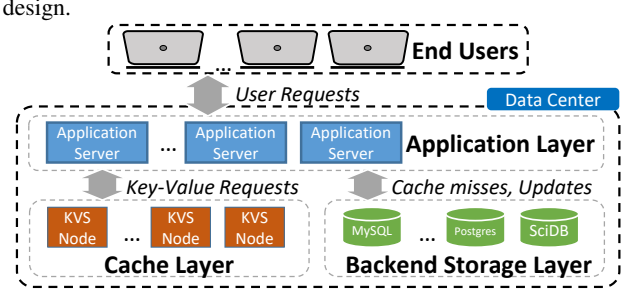


Figure 1: Using key-value stores as caching layer

In theory, one can scale up the KVS cluster by increasing the total amount of RAM or the number of servers in the KVS cluster. In practice, however, hardware cost, power/thermal concerns and floor space can become obstacles to scaling the memory pool size. As illustrated in Table 1, NAND flash is 10X cheaper, consumes 10-100X less power and offers two orders of magnitude greater storage density over DRAM. These characteristics make flash-based KVS a viable alternative for scaling up the KVS cache size. A downside of flash-based KVS is that flash has significantly higher latency than DRAM. Typically flash memory has read latency of 100μs and write latency of milliseconds. DRAM, on the other hand, offers latency of 10-20ns, which is more than four orders of magnitude better than flash. Thus to realize the advantage of flash, a flash-based architecture must overcome this enormous latency differential. One silver lining is that many applications can tolerate millisecond latency in responses. Facebook's memcached cluster reports 95th percentile latency of 1.135 ms [52]. Netflix's EVCache KVS cluster has 99th percentile latency of 20 ms and is still able to deliver rich experience for its end users [3].

Table 1: Storage device comparison as in 2016, prices are representative numbers on the Internet

Device	Storage Technology	Capacity *(GB)*	Bandwidth *(GBps)*	Power *(Watt)*	Price *(USD)*	Price/Capacity *(USD/GB)*	Power/Capacity *(mWatt/GB)*
Samsung 950 PRO Series [7]	3D V-NAND PCIe NVMe	512	2.5	5.7	318.59	0.62	11.13
Samsung 850 PRO Series [6]	3D V-NAND SATA SSD	2000	0.5	3.5	836.51	0.42	1.65
Samsung M393A2G40DB0-CPB [8, 9]	DDR4-2133MHz ECC RDIMM	16	17	3.52	89.99	5.62	220

This paper presents BlueCache, a new flash-based architecture for KVS clusters. BlueCache uses hardware accelerators to speed up KVS operations and manages communications between KVS nodes completely in hardware. It employs several technical innovations to fully exploit flash bandwidth and to overcome flash's long latencies: 1) Hardware-assisted auto-batching of KVS requests; 2) In-storage hardware-managed network, with dynamic allocation of dedicated virtual channels for different applications; 3) Hardware-optimized set-associative KV-index cache; and 4) Elimination of flash translation layer (FTL) with a log-structured KVS flash manager, which implements simple garbage collection and schedules out-of-order flash requests to maximize parallelism.

Our prototype implementation of BlueCache supports 75X more bytes per watt than the DRAM-based KVS and shows:

- 4.18X higher throughput and 4.68X lower latency than the software implementation of a flash-based KVS, such as Fatcache [63].
- Superior performance than memcached if capacity cache misses are taken into account. For example, for applications with the average query size of 1KB, GET/PUT ratio of 99.9%, BlueCache can outperform memcached when the latter has more than 7.4% misses.

We also offer preliminary evidence that the BlueCache solution is scalable: a four-node prototype uses 97.8% of the flash bandwidth. A production version of BlueCache can easily support 8TB of flash per node with 2.5 million requests per second (MRPS) for 8B to 8KB values. In comparison to an x86-based KVS with 256GB of DRAM, BlueCache provides 32X larger capacity with 1/8 power consumption. It is difficult to assess the relative performance of the DRAM-based KVS because it crucially depends on the assumptions about cache miss rate, which in turn depends upon the average value size. BlueCache presents an attractive point in the cost-performance trade-off for data-center-scale key-value caches for many applications.

Even though this paper is about KVS caches and does not exploit the persistence of flash storage, the solution presented can be extended easily to design a persistent KVS.

Paper organization: In Section 2 we discuss work related to KVS caches. In Section 3 we describe the architecture of BlueCache, and in Section 4 we describe the software library to access BlueCache hardware. In Section 5 we describe a hardware implementation of BlueCache, and show our results from the implementation in Section 6. Section 7 concludes the paper.

2. RELATED WORK

We will first present the basic operations of KVS caches and discuss the related work for both DRAM-based and flash-based KVS.

2.1 A Use Case of Key-Value Store

A common use case of KVSs is to provide a fast look-aside cache of frequently accessed queries by web applications (See Figure 1). KVS provides primitive hash-table-like operations, SET, GET and DELETE on *key-value* pairs, as well as other more complex operations built on top of them. To use KVS as a cache, the application server transforms a user read-request into multiple GET requests, and checks if data exists in KVS. If there is a cache hit, the application server collects the data returned from KVS and formats it as a response to the end user. If there is a cache miss, application server queries the backend storage for data, and then issues a SET request to refill the KVS with the missing data. A user write-request, *e.g.* a Facebook user's "unfriend" request, is transformed by the application server into DELETE requests for the relevant key-value pairs in the KVS, and the new data is sent to the backend storage. The next GET request for the updated data automatically results in a miss which forces a cache refill. In real-world applications, more than 90% KVS queries are GETs [11, 18].

Application servers usually employ hashing to ensure load balancing across KVS nodes. Inside the KVS node, another level of hashing is performed to compute the internal memory address of the key-value pairs. KVS nodes do not communicate with each other, as each is responsible for its own independent range of keys.

2.2 DRAM-based Key-Value Store

DRAM-based key-value stores are ubiquitous as caching solutions in data centers. Like RAMCloud [54], hundreds to thousands of such KVS nodes are clustered togother to provide a distributed in-memory hash table over fast network. Figure 2 shows a typical DRAM-based KVS node. Key-value data structures are stored on KVS server's main memory, and external clients communicate with the KVS server over network interface card (NIC). Since both keys and values can be of arbitrary size, an KV-index cache for each key-value pair is kept in a separate data structure. In the KV-index cache the data-part of the key-value pair just contains a pointer to the data which resides in some other part of the cache. This index data structure is accessed by hashing the key generated by the application server. If the key is found in the KV-index cache, the data is accessed in the DRAM by following the pointer in the KV-index cache. Otherwise, the KV-index cache and the DRAM have to be updated by accessing the backend storage.

KVS caches have two important software components: 1) the network stack that processes network packets and injects them into CPU cores; 2) the key-value data access that reads/writes key-value data structures in main-memory. These two components have a strong producer-consumer dataflow relationship, and a lot of work has been done to optimize each component as well as the communication between them.

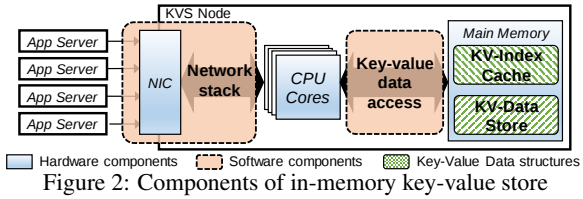


Figure 2: Components of in-memory key-value store

A past study [58] has found that more than 90% processing time in memcached is spent in the OS kernel network processing. Researchers have demonstrated as much as 10X performance improvement by having user-space network stack [62, 31, 32, 51, 23, 36, 47, 45]. For example, Jose et al. [31, 32] investigated the use of RDMA-based communication over InfiniBand QDR network, which reduced KVS process latency to below 12μs and enhanced throughput to 1.8MRPS. Another example, MICA [47, 45], exploits modern NIC features such as multiple queues and flow-steering features to distribute packets to different CPU cores. It also exploits Intel Data Direct I/O Technology (DDIO) [29] and open-source

driver DPDK [30] to let NICs directly inject packets into processors' LLC, bypassing the main memory. MICA shards/partitions key-value data on DRAM and allocates a single core to each partition. Thus, a core can access its own partition in parallel with other cores, with minimal need for locking. MICA relies on software prefetch for both packets and KVS data structures to reduce latency and keep up with high speed network. With such a holistic approach, a single MICA node has shown 120MRPS throughput with 95th percentile latency of 96μs [45].

Berezecki et al. [15] use a 64-core Tilera processor (TILEPro64) to run memcached, and show that a tuned version of memached on TILEPro64 can yield at least 67% higher throughput than low-power x86 servers at comparable latency. It showed 4 TILEPro64 processors running at 866Mhz can achieve 1.34MRPS. This approache offers less satisfactory improvements than x86-based optimizations.

Heterogeneous CPU-GPU KVS architectures have also been explored [64, 27, 28]. In particular, Mega-KV [64] stores KV-index cache on GPUs' DRAM, and exploits GPUs' parallel processing cores and massive memory bandwidth to accelerate the KV-index cache accesses. KV-data store is kept separately on the server DRAM, and network packets are processed, like in MICA, using Intel's high-speed I/O [30, 29]. On a commodity PC with two Nvidia GTX 780 GPUs and two CPUs, Mega-KV can process up to 166 MRPS with 95th percentile latency of 410μs [64].

Researchers have also used FPGAs to offload parts [48, 40, 24] or the entirety [19, 16] of KVS, and demonstrated good performance with great power efficiency. Xilinx's KVS has the highest performance based on this approach, and achieves up to 13.2MRPS by saturating one 10GbE port [16], with the round-trip latency of 3.5μs to 4.5μs. It also shows more than 10x energy efficiency compared with commodity servers running stock memcached.

Discussion: An insightful performance comparison of different approaches is difficult. First of all, all performance results are reported assuming no *capacity* cache misses. Even cache misses due to updates are not properly described. For example, both MICA and Mega-KV experiments that showed the highest performance (>120MRPS) assumed 8B key and 8B value. If the application had 1KB values, then the same cache will hold 128X fewer objects, which should significantly increase the number of capacity misses. Unfortunately, the capacity misses vary from application to application and cannot be estimated based of the number of objects in KVS. Even if capacity misses are rare, the throughput in terms of MRPS will be much lower for larger values. For example, MICA performance drops from 5MRPS to .538MRPS per core if the value size is increased from 8B to 1KB [47]. Moreover, a simple calculation shows that for 1KB values, one 10Gbps port cannot support more than 1.25MRPS.

The second point to note is that higher throughput requires more hardware resources. Mega-KV shows 1.38X more performance than MICA but also increased power consumption by 2X. This is because it uses two Nvidia GTX 780 GPUs which consume approximately 500W [4]. If we look at the performance normalized by the number and speed of network ports then we will reach a different conclusion. For example, MICA shows a performance of 120MRPS using 12 10Gbps Ethernet ports (10MRPS per port), while Xilinx [16] achieves 13.2MRPS using only one 10Gbps port.

One can also ask the question exactly how many resources are needed to keep a 10Gbps port busy. MICA experiments show that 2 Xeon cores are enough to keep up with a 10Gbps Ethernet port [45], and Xilinx experiments show that an FPGA-based implementation can also easily keep up with the 10Gbps port [16]. A past study has pointed out that traditional super-scalar CPU core pipeline of x86 can be grossly underutilized in performing the required KVS computations (networking, hashing and accessing key-value pairs) [48]. The last level data cache, which takes as much as half of the processor area, can be ineffective [48] and can waste a considerable amount of energy. A simple calculation in terms of cache needed to hold all in-flight requests and network packets shows that MICA's 120MRPS throughput can be sustained with only 3.3 MB of LLC, while a dual-socket Intel Xeon processor has a 60MB LLC!

2.3 Flash-based Key-Value Store

There have been several efforts to use NAND flash in KVS designs because it provides much higher storage density, lower power per GB and higher GB per dollar than DRAM [61, 14, 50, 44, 57, 49]. In one organization, NAND flash is used as a simple swap device [60, 39] for DRAM. However, the virtual memory management of the existing OS kernels is not suitable for NAND flash because it leads to excessive read/write traffic. NAND flash is undesirable for small random writes because an entire page has to be erased before new data is appended. High write-traffic not only wastes the I/O bandwidth but shortens the NAND lifetime [12].

A better flash-based KVS architecture uses flash as a cache for objects where the object granularity is one page or larger [46, 25]. Examples of such an architecture include Twitter's Fatcache [63], FAWN-KV [10], Hybrid Memory [56], Xilinx's KVS [17] and FlashStore [22]. KV-index cache stores key-value metadata such as timestamps, which has frequent updates. Like Figure 2, they move key-value data to flash while keeping KV-index cache in DRAM, because in-place index updates on flash would make it prohibitively inefficient and complicated. The NAND flash chips are written as a sequence of blocks produced by a log-structured flash management layer to overcome NAND flash's overwriting limitations [43, 42].

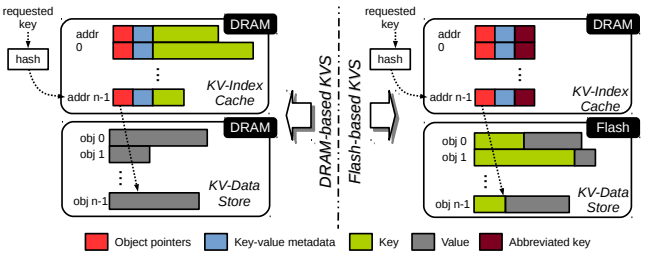


Figure 3: Internal data structures of DRAM-based KVS and flash-based KVS

A superior way of maintaining an in-memory KV-index cache is to use fixed-size representation for variable size keys (See Figure 3). This can be done by applying a hash function like SHA-1 to keys [63, 22] and keeping the fixed-size abbreviated keys on DRAM with pointers to the full keys and values on flash. False positive match of an abbreviated key can be detected by comparing request key with the full key stored on flash. Such data structure organization ensures that 1) on a KV-index cache miss, there is no flash access; and 2) on a KV-index cache hit, a single flash access is needed to access both the key and data.

One can calculate the rarity of a false positive match of an abbreviated key. Assuming each hash bucket has four 128-bit entries, an 8GB hash table has $2^{33-2-4} = 2^{27}$ hash buckets. Assuming 21-bit abbreviated keys, the false positive for key hit on a index entry is as low as $1/(2^{27+21}) = 1/2^{48}$. This ensures that a negligible portion of KVS misses are penalized by expensive flash reads.

One can estimate the size of KV-index cache based on the average size of objects in the workload. For example, assuming 16-byte index entries, a 16GB KV-index cache can store 2^{30} or ~a billion key-value pairs in flash. Such an index cache can address 1TB

KV-data store, assuming average object size of 1KB, or can address 250GB KV-data store, assuming average object size of 256B. Thus, in this system organization, a terabyte of SSD can be paired with 10 to 50 GB of DRAM to increase the size of KVS by 10X to 100X on a single server.

KV-index cache reduces write traffic to NAND flash by 81% [56], which implies 5.3X improvement in storage lifetime. FlashStore [22] is the best performing single-node flash-backed KVS in this category, achieving 57.2 KRPS. Though the capacity is much larger, the total performance, ignoring DRAM cache misses, is more than one to two orders of magnitude less than the DRAM-based solutions. Unlike MICA [45], the throughput of FlashStore is limited by the bandwidth of the flash device and not the NIC. If the flash device organizes NAND chips into more parallel buses, the throughput of flash-based KVS will increase, and it should be possible for the KVS to saturate a 10Gbps Ethernet. The throughput can be increased even further by eliminating flash translation layer (FTL) [21] in SSDs, as has been shown for flash-based filesystems, such as SDF [55], F2FS [41], REDO [42] and AMF [43].

3. BLUECACHE ARCHITECTURE

The BlueCache architecture consists of a homogeneous array of hardware accelerators which directly manage key-value pairs on error-corrected NAND-flash array of chips without any general purpose processor (See Figure 4). It organizes key-value data structures the same way as shown in Figure 3. The KV-index cache stores abbreviated keys and key-value pointers on DRAM, and the KV data, i.e., the full key and value pairs, on flash. BlueCache also uses a small portion of DRAM to cache hot KV-data-store entries. Based on the average size of key-value pairs, BlueCache architecture typically requires the DRAM capacity to be 1/100 to 1/10 of the flash capacity in order to address all the data on flash.

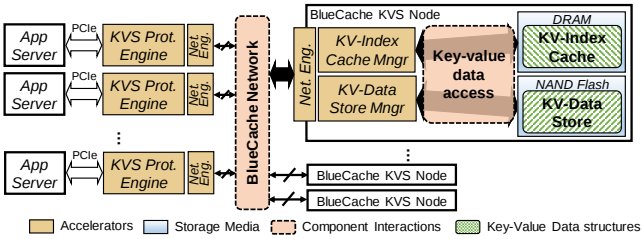


Figure 4: BlueCache high-level system architecture

The *KVS Protocol Engine* is plugged into each application server's PCIe slot and forwards each KVS request to the BlueCache KVS node responsible for processing it. In order to maximize parallelism and maintain high performance, KVS requests are batched on application server's DRAM and transferred to the Protocol Engine as DMA bursts. When responses are returned from various KVS nodes, the Protocol Engine batches the responses and sends them back to the application server. The Protocol Engine communicates with KVS nodes via high-speed *BlueCache network* accessed using the *Network Engine*; there is no communication between KVS nodes.

Each KVS node (See Figure 4) runs a *KV-Index Cache Manager* and a *KV-Data Store Manager* to manage the KV-index cache and the KV-data store, respectively. Based on the size of KV-index cache, some DRAM is also used to cache KV-data store. Just like the KVS cluster described in Section 2.1, BlueCache architecture deploys two levels of hashing to process a key-value query. The first level of hashing is performed by the KVS Protocol Engine to find the KVS node responsible for the query, while the second level of hashing is performed within the KVS node to find the corresponding index entry in the KV-index cache.

We use Field-programmable Gate Arrays (FPGA), which contain reprogramable logic blocks, to implement BlueCache (See Section 5). In our FPGA-based implementation, each FPGA board has its own DRAM and NAND flash chips and is directly plugged into a PCI Express slot of the application server (see Figure 5). Both the KVS Protocol Engine and the KVS node are mapped into a single FPGA board and share the same Network Engine. FPGA boards can communicate with each other using 8 10Gbps bidirectional serial links [35]. In the following sections, we describe each hardware accelerator in detail.

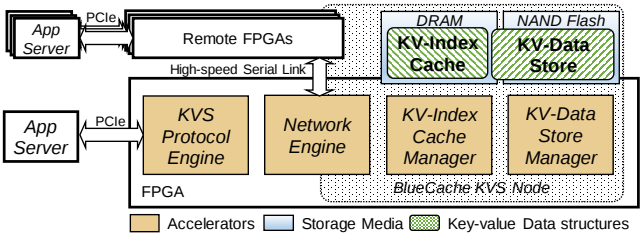


Figure 5: A BlueCache architecture implementation

3.1 KVS Protocol Engine

Unlike KVS such as memcached [52] which relies on client software (*e.g.* getMulti(String[] keys)) to batch KVS queries, Bluecache uses hardware accelerators to automatically batch KVS requests from multiple applications. The application server collects KVS requests in 8KB segments in the DRAM and, when a segment is filled up, passes the segment ID to the Protocol Engine for a DMA transfer (See Figure 6). The application server then picks up a new free segment and repeats the above process. The Protocol Engine receives requests in order and sends an acknowledgement to the application server after reading the segment. A segment can be reused by the application server after the acknowledgement has been received. Since KVS requests are of variable length, a request can be misaligned with the segment boundary. In such cases, the Protocol Engine merges segments to produce a complete KVS request. KVS responses are batched together similarly by the Protocol Engine and sent as DMA bursts to the application server. Our implementation uses 128 segments DMA buffers in each direction.

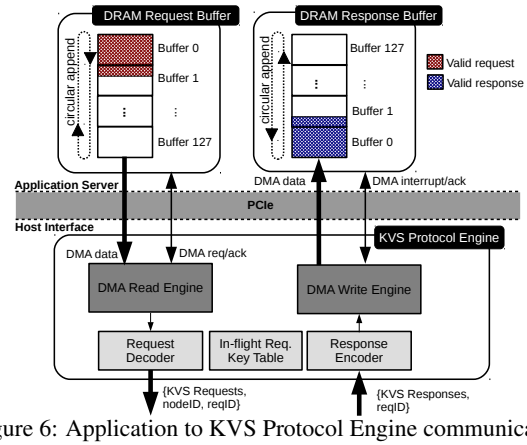


Figure 6: Application to KVS Protocol Engine communication

The *request decoder* (See Figure 6) computes the destination BlueCache node ID by using the equation:

$$nodeID = hash(key) \bmod \#nodes \quad (1)$$

to distribute the requests across the KVS nodes evenly. The hash function in the equation should be a *consistent hashing function* [37], so that when a node joins or leaves the cluster, minimum number of key-value pairs have to be reshuffled. KVS Protocol Engine also keeps a table of the in-flight request keys because the responses do

not necessarily came back in order. In Section 3.3 we will describe how request and response keys are matched to form a response for the application server.

3.2 Network Engine

Each request carries a destination node ID which is used by the Network Engine to route the requests to its destination. Each request also carries the source ID, *i.e.*, the ID of the node that sends the request; these are used by the Network Engine to send back responses. The Network Engine splits the KVS requests into local and remote request queues (Figure 7). The requests in the remote-request queue are forwarded to remote node via the network router. When a remote request is received at its destination node, it is merged into the local-request queue of the remote node. Similarly, the responses are split into the local-response queue and the remote-response queue depending on their requests' origins. The response network router forwards the non-local KVS responses to the remote nodes, which are later merged into the local response queues at their sources. The request and response networks are kept separate using the support for virtual networks in our network implementation [35]. With the support of virtual networks, Bluecache can also dynamically assign a dedicated network channel for a running application, so that the network interference between applications is minized.

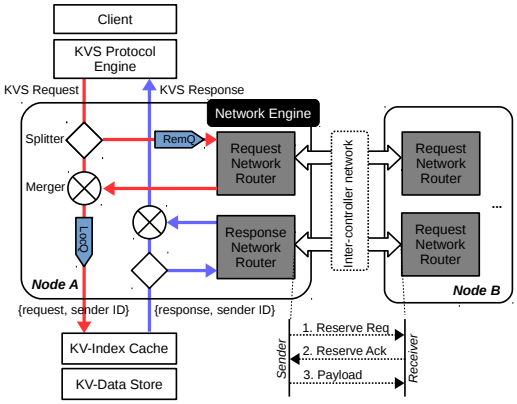


Figure 7: Network Engine Architecture

Our multi-FPGA platforms network is designed to be lossless and thus, uses a simple *handshake* network protocol. Before sending a payload, the sender sends a reserve request to the receiver with the size of the payload. The receiver then reserves the memory for the incoming payload, and acknowledges the sender. After receiving the acknowledgment, the sender sends the payload data to the receiver. Because of the simplicity of the protocol, our network has only a 1.5μs latency per hop.

3.3 KV-Index Cache Manager

The KV-index cache is organized as a *set-associative* hash table on DRAM (Figure 8). A key is mapped into a hash bucket by computing the *Jenkins hash function* [1], which is also used in memcached. A hash bucket contains fixed number of 128-bit index entries, each of which contain 5 fields describing a key-value pair. A 41-bit *key-value pointer* field points to the location of the key-value pair in a 2TB address space; a 8-bit *key length* field represents the key size (up to 255B); a 20-bit *value length* field represents value size (up to 1MB); a 32-bit *timestamp* field represents the key-value pair's latest access time, which is used to evict an index entry; and a 21-bit field stores the truncated SHA-1 hash value of the key to resolve hash function collisions. The maximum key and value sizes are chosen as in memcached. If there is a need to support a larger key or value size, the index entry has to be changed by either increasing the size of the index entry, or decreasing the abbreviated key field size within an acceptable false positive hit rate.

Since it is inefficient to use linked list or rehashing to handle hash function collisions in hardware, we check all four index entries in the hash bucket *in parallel*. For a KV-index cache hit, the requested key needs to match the abbreviated key and the key length.

Because our FPGA platform returns 512-bit data per DRAM request, we assign each hash bucket to every 512-bit aligned DRAM address. In this way, each hash bucket has four index entries. Yet, such a 4-way set-associative hash table design can guarantee good performance since each hash table operation only needs one DRAM read to check keys, and one DRAM write to update timestamps and/or insert a new entry. Higher set associativity can reduce conflict misses of KV-index cache and increase performance in theory. Yet, increasing associativity requires more hardware, more DRAM requests and more clock cycles per operation, which can offset benefits of a higher hit rate.

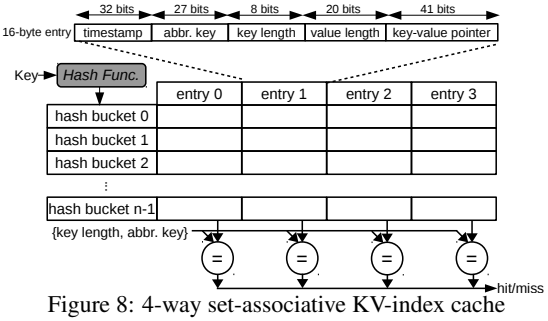


Figure 8: 4-way set-associative KV-index cache

When there are more than four collisions at a hash bucket, an old index entry is evicted. Since KVS workload shows strong temporal locality [11], the KV-Index Cache Manager uses LRU replacement policy to select the victim by reading their timestamps. When a key-value pair is deleted, only its index entry is removed from KV-index cache, and the object on KV-data store is simply ignored, which can be garbage collected later.

3.4 KV-Data Store Manager

The key-value object data is stored either in the DRAM or in the flash (Figure 9). Key-value pairs are first written in DRAM and when it becomes full, less popular key-value pairs are evicted to the flash to make space. Each entry in the KV-data store keeps a backward pointer to the corresponding entry in the KV-index cache. When the KV data is moved from the DRAM to the flash, the corresponding KV index entry is updated. The most significant bit of key-value pair pointer in the KV-index cache indicates whether the data is in DRAM or flash.

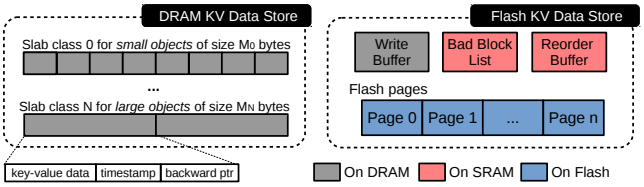


Figure 9: KV-data store architecture

DRAM KV-Data Store: The DRAM KV-data store is organized as a collection of slab classes, where each slab class stores objects of a predetermined size (Figure 9). A slab-structured DRAM store implementation requires simple hardware logic, and its dense format enables better RAM efficiency. The KV-data store in memcached [52] uses a similar technique in order to minimize DRAM fragmentation in software. When an object needs to be moved to the flash, the victim is determined by the entry with the oldest

timestamps amongst four randomly-chosen entries in the slab. After the victim is evicted, the in-memory KV-index cache is updated using backward pointer of the evicted object. This policy behaves like a pseudo-LRU replacement policy, which keeps hot objects in DRAM. Since the KV-Data Store shares the DRAM with the KV-index cache, and the size of DRAM KV-data store can be dynamically adjusted in respect to the size of active KV-index cache.

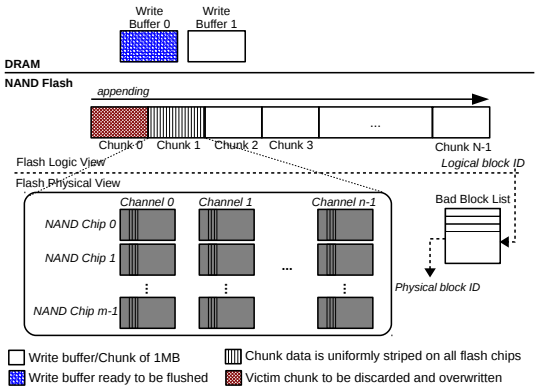


Figure 10: Log-structured flash store architecture

Flash KV-Data Store: The data structure on the KV flash data store is very different from the DRAM KV-data store. On flash one has to erase a page before it can be overwritten. Though flash chip allows reads and writes at the granularity of a page, it permits only block erasures, where the typical size of a block is 64 to 128 pages. Block erasure has high overhead (several milliseconds) and is performed in the background as needed. To avoid small overwrites, we use log-structured techniques [43, 42] for writing in flash. Victims from the DRAM KV-data store are collected in a separate DRAM buffer, and written to flash in 128-page chunks as a log (Figure 9, Figure 10). The KV-data manager has the flexibility to map the data pages on the flash array and instead writing a 128-page chunk on one chip, it stripes the chunk across N flash chips for maximum parallelism. This also ensures good wear-leveling of flash blocks. The manager also has to maintain a list of bad blocks, because flash blocks wear out with usage (Figure 10).

Similar to work [20], the KV-Data Store Manager schedules reads concurrently to maximize parallelism, and consequently, the responses do not necessarily come back in order. A reorder buffer is used to assemble pages to construct the response for a request [49].

Flash KV-data store also implements a minimalist garbage collection algorithm for KV-data store. When space is needed, an old flash chunk from the beginning of the log is simply erased and overwritten. The keys corresponding to the erased data have to be deleted from the KV-index cache to produce a miss. In KVS workloads, newly written data has higher popularity [11] and by overwriting the oldest data, it is more likely that cold objects are replaced by hot ones. Consequently new objects are always written in the DRAM cache. BlueCache's simple garbage collection ensures good temporal locality.

Such a KV-data store requires direct accesses to NAND flash chips, which is very difficult to implement with commercial SSDs. Manufactures often deploy a flash translation layer (FTL) inside SSDs to emulate hard drives behaviors, and this comes in the way of exploiting the full flash bandwidth [55, 17]. The BlueCache architecture eliminates FTL and uses direct management of parallel NAND flash chips to support KVS operations.

3.5 From KVS Cache to Persistent KVS

The BlueCache architecture described so far does not use the non-volatility of flash it only exploits the capacity advantage of flash over DRAM. However, it can be easily transformed into a persistent KVS.

In order to guarantee persistency, BlueCache needs to recover from crashes due to power outage as well as other system failures. If the KV-index cache is lost then the KV-data store essentially becomes inaccessible. Like filesystems [43, 41], we can periodically write the KV-index cache to flash for persistency. The KV-index cache checkpoints are stored in a different region of the flash than the one hold the KV data. In this way, a slightly older version of KV-index cache can be brought back in DRAM quickly. We can read the flash KV-data store, replay log from the timestamp of the KV-index cache checkpoint, and apply SET operation on the key-value pairs sequentially from the log to recover the KV-index cache. When checkpoints are being made, updates to KV-index cache should not be allowed. To allow high availability of the KVS, we could devise a small interim KV hash table, which can be later merged with KV-index cache and KV-data store after the checkpoint operation finishes.

The write policy for the DRAM data cache should depend on the requirement of data recovery by different applications. For applications that need fast retrieval of hot data, write-through policy should be chosen because key-value pairs will be immediately logged on to flash. Yet, this comes at the cost of having a smaller effective KVS size. On the other hand, write-back policy works for applications which have relaxed requirement for recovering the most recent data and benefit from larger capacity.

4. SOFTWARE INTERFACE

In data centers, a KVS cluster is typically shared by multiple application. BlueCache provides a software interface, which allows many multi-threaded applications to share the KVS concurrently. The software interface implements a partial memcached client API consisting of three basic C++ functions: GET, SET, DELETE, as listed below. The C++ functions can be also accessed by other programming languages via their C wrappers, such as JAVA through Java Native Interface (JNI).

```
1 bool bluecache_set(char* key, char* value,
2     size_t key_length, size_t value_length);
3 void bluecache_get(char* key, size_t key_length,
4     char** value, size_t* value_length);
5 bool bluecache_delete(char* key, size_t key_length);
```

These functions provides *synchronous* interfaces. The BlueCache KVS throughput via the software interface increases as there are more concurrent accesses, and it stops scaling beyond 128 concurrent application threads (See Section 6.3).

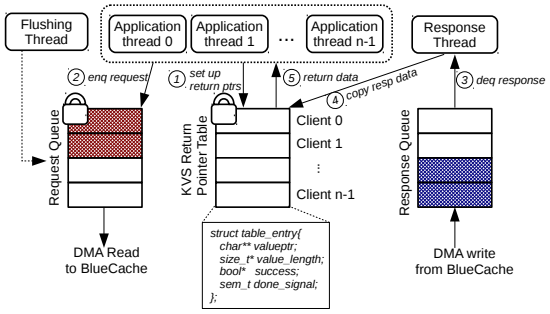


Figure 11: BlueCache software interface

BlueCache software interface is implemented by using three types of threads (Figure 11). *Application threads* send KVS queries via the API. *Flushing thread* periodically pushes partially filled DMA segments to host interface in background if there are no incoming

requests. *Response thread* handles DMA interrupts from hardware, and dispatches responses to the application threads.

Furthermore, BlueCache software interface has three data structures owned by different threads. The *request queue* buffers all KVS requests, and is shared by the application threads and the flushing thread. The *response queue* buffers all KVS responses returned from hardware, which is solely owned by the response thread. The *KVS return pointer table* is shared by the application threads and the response threads, and maintains an array of return pointers for each clients. Once the hardware returns KVS query results, the response thread can signal the right application thread with the data. All the shared data structures are protected by mutex locks.

Figure 11 also illustrates the internal operations of the software interface. When an application thread queries BlueCache, it first sets up its return pointers on the KVS return pointer table. Second, the application thread push the KVS queries to the request queue, which is later send to BlueCache via DMA. The client thread then waits for the response. The response thread receives an interrupt from the hardware after BlueCache pushes KVS query results to the response queue. The response thread then dequeues the KVS response, copies response data into a byte array, and signals the application thread with the response by referring to the information on the KVS return pointer table.

5. BLUECACHE IMPLEMENTATION

We use Field Programmable Gate Arrays (FPGA) to implement BlueCache. In this section we describe BlueCache's implementation platform, hardware resource usage, and power characteristics in order.

5.1 BlueDBM Platform

BlueCache is implemented on MIT's BlueDBM [34, 33] cluster, a multi-FPGA platform consisting of identical nodes. Each node is a Intel Xeon server with a BlueDBM storage node plugged into a PCIe slot. Each BlueDBM node storage consists of a Xilinx VC707 FPGA development board with two 0.5TB custom flash cards. The VC707 board is the primary carrier card, and it has a Virtex-7 FPGA and a 1GB DDR3 SODIMM. Each flash card is plugged into a standard FPGA Mezzanine Card (FMC) port on the VC707 board, and provides an error-free parallel access into an array NAND flash chips (1.2GB/s or 150K IOPs for random 8KB page read) [49]. Each BlueDBM storage node has four 10Gbps serial transceivers configured as Aurora 64B/66B encoded links with 0.5μs latency per hop. BlueDBM also supports a virtual network over the serial links, which provides virtual channels with end-to-end flow control [35].

BlueCache KVS components are mapped into different BlueDBM parts. The Xeon computer nodes are used as application servers, and BlueDBM storage nodes are used as the KVS. On each BlueDBM storage node, the KV-index cache is stored in the DDR3 SODIMM, and the KV-data store is stored in the flash. The high-speed serial links are used as the BlueCache network. The Virtex-7 FPGA hosts all the hardware accelerators, which are developed in the high-level hardware description language Bluespec [2].

On the BlueDBM's Xeon servers, the software interface is developed on Ubuntu 12.04 with Linux 3.13.0 kernel. We used the Connectal [38] hardware-software codesign library which provides RPC-like interfaces and DMA over PCIe. We used a PCIe gen 1 version of Connectal, which supports 1 GB/s to and 1.6GB/s from FPGA. Two flash cards would provide a total of 2.4 GB/s bandwidth which would exceed the PCIe gen 1 speed; however, only one flash card is used due the limitation of FPGA resources to be explained in Section 5.2.

5.2 FPGA Resource Utilization

Table 2: Host Virtex 7 resource usage

Module Name	LUTs	Registers	RAMB36	RAMB18
KVS Protocol Engine	17128	13477	8	0
Network Engine	74968	184926	189	11
KV Data Store Manager	33454	32373	228	2
KV Index Cache Manager Table	52920	49122	0	0
Virtex-7 Total	265660 (88%)	227662 (37%)	524 (51%)	25 (1%)

Table 2 shows the VC707 resource usage for a single node of a four node configuration with one active flash card per node. This configuration uses most (88%) of the LUTs and half the BRAM blocks. Given the current design, the VC707 does not support a full-scale BlueCache KVS by using all the hardware resources provided by BlueDBM platform (20-node KVS node with 20TB flash capacity).

5.3 Power Consumption

Table 3: BlueCache power consumption vs. other KVS platforms

Platforms	Capacity *(GB)*	Power *(Watt)*	Capacity/Watt *(GB/Watt)*
FPGA with Raw Flash Memory(BlueCache)	20,000	800	25.00
FPGA with SATA SSD(memcached) [17]	272	27.2	10.00
Xeon Server(FlashStore) [22]	80	83.5	0.96
Xeon Server(optimized MICA) [45]	128	399.2	0.32
Xeon Server+GPU(Mega-KV) [64]	128	899.2	0.14

Table 3 compares the power consumption of BlueCache with other KVS systems. Thanks to the lower consumption of FPGAs, one BlueCache node only consumes approximately 40 Watts at peak. A 20-node BlueCache cluster consumes 800 Watts and provides 20TB of key-value capacity. Compared to other top KVS platforms in literature, BlueCache has the highest capacity per watt, which is at least 25X better than x86 Xeon server platforms, and 2.5X over an FPGA-based KVS with SATA SSDs. A production system of BlueCache can easily support 8TB NAND flash chips *per node*, and a single node can provide 2.5M IOPs of random 8KB page read and comsumes less than 50W (1/8 of a Xeon server) [59].

6. EVALUATION

6.1 Single-Node Performance

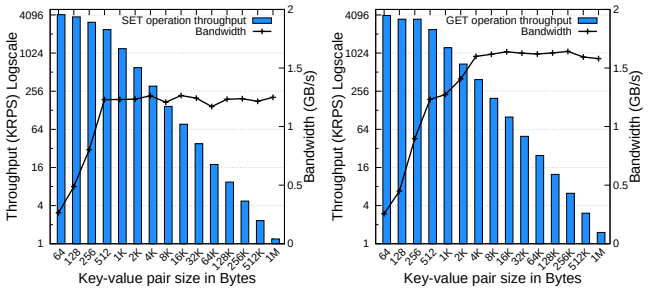
We evaluated GET and SET operation performance on a single BlueCache node. We measured both throughput and latency of the operations. Measurements are made from application servers without multi-thread software interface to test the peak performance of the BlueCache hardware.

6.1.1 DRAM-only Performance

This part evaluates the performance of a single-node BlueCache DRAM-only implementation. All key-value pairs are resident on the slab-structured DRAM store of the KV data cache. Measurements are made with key-value pairs of different sizes, keys are all 32B.

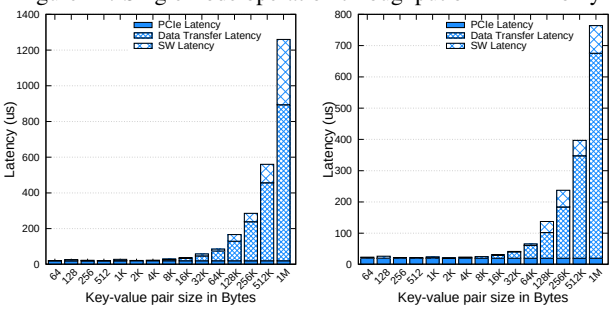
1) Operation Throughput: Figure 12 shows the throughput of a single-node DRAM-only BlueCache. DRAM-only BlueCache has peak performance of 4.14MRPS for SET operations and 4.01MRPS for GET. This is a 10X improvement over the stock memcached running on a Xeon server which support 410 KRPS at peak.

BlueCache operation throughput is limited by different hardware components depending on the size of key-value pairs. For small key-value pairs (<512B), operation throughput is bottlenecked by



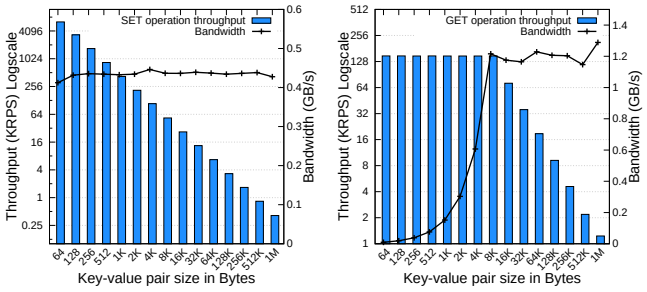
(a) SET Throughput (b) GET Throughput

Figure 12: Single-node operation throughput on DRAM only



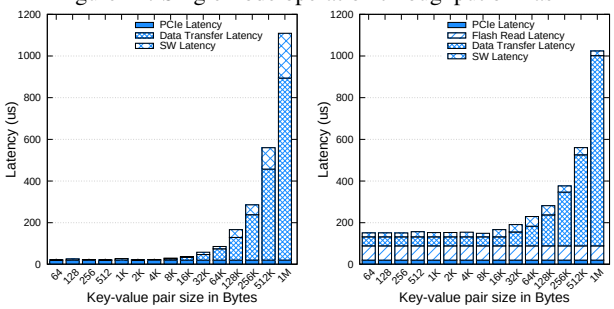
(a) SET Latency (b) GET Latency

Figure 13: Single-node operation latency on DRAM only



(a) SET Throughput (b) GET Throughput

Figure 14: Single-node operation throughput on flash



(a) SET Latency (b) GET Latency

Figure 15: Single-node operation latency on flash

random access bandwidth of the DDR3 memory. DRAMs support fast sequential accesses, but is slow for random accesses. As measured from the 1GB DDR3 DRAM, bandwidth is 12.8GB/s for sequential access vs 1.28GB/s for random access. For large key-value pairs (>512B), DRAM sequential accesses dominate but PCIe bandwidth limits the performance of BlueCache. PCIe gen1 limits transfers from the application server to BlueCache at 1.2GB/s for SETs and 1.6GB/s in the reverse direction for GETs.

2) Operation Latency: Operation latency consists of PCIe latency, data transfer latency and software latency. Figure 13 shows that operation latency of single-node DRAM-only BlueCache varies from 20μs to 1200μs. PCIe latency is constant about 20μs (10μs per direction), and it is the dominant latency source when key-value pairs are small(<8KB). Data transfer latency increases as key-value pairs become larger, and it dominates latency for key-value pairs larger than 8KB. Software latency includes processing time of PCIe interrupts, DMA requests and other software components. Software latency is small because only one interrupt is needed for key-value sizes smaller than 8KB. For bigger key-value pairs, there are more DMA bursts which requires more interrupts per KVS request, and the software overhead becomes significant.

6.1.2 *Performance with Flash*

This section evaluates the performance of a single-node BlueCache implementation with flash. All key-value pairs are resident on the log-structured flash store of the key-value data cache. Measurements are made with key-value pairs of different sizes, keys are all 32B.

1) Operation Throughput: Figure 14 shows the throughput of a single-node BlueCache with one flash board. Using flash, BlueCache has peak performance of 6.45MRPS for SET operations and 148.49KRPS for GET operations with 64 byte key-value pairs.

For SET operations, key-value pairs are buffered in DRAM and then written to flash in bulk at the 430MB/s NAND Flash write bandwidth (Figure 14(a)). For 64-byte key-value pairs, SET operations on flash has more bandwidth than DRAM (Figure 14(a) vs. Figure 12(a)). The slower performance on DRAM is due to poor random accesses on the single-channel DDR3 SDRAM. For flash, performance is better because there are more sequential DRAM write pattern from the DRAM write buffer on the flash KV data store.

For GET operations, the requirement to read 8KB NAND pages limits throughput to 148K IO/s. As shown in Figure 14(a), for key-value pairs smaller than 8KB, only one page of flash is read and so the operation throughput is same as that of flash (~148KPS). For key-value pairs greater than 8KB, multiple pages are read from flash, and operation throughput is limited by flash read bandwidth.

2) Operation Latency: Figure 15 shows operation latency of a single-node BlueCache. SET operation latency of BlueCache using flash varies from 21μs to 1100μs, which is similar to that using DRAM only, because all SET operations are buffered in DRAM before being written to flash. GET operation latency consists of PCIe latency, flash read latency, data transfer latency, and software latency. GET operation latency varies from 150μ to 1024μs. Data transfer latency dominates, because an entire flash page needs to be fetched even for small reads.

6.2 Multi-Node Performance

We measured multi-node performance by chaining four BlueCache nodes together in a linear array. Each BlueCache node is attached to an application server via PCIe.

1) Operation Throughput: We measured BlueCache throughput under the following scenarios: (1) a single application server accessing multiple BlueCache nodes (Figure 16(a)). (2) multiple application servers accessing a single BlueCache node (Figure 16(b)). (3) multiple application servers accessing multiple BlueCache nodes (Figure 16(c)). All accesses are 40,000 random GET operations of 8KB key-value pairs on flash.

The first scenario examines the scalability of BlueCache when there is only one application server accessing BlueCache. We observed some speed-up (from 148 KPRS to 200 KRPS) by accessing multiple storage nodes in parallel (See Figure 16(a)), but ultimately we are bottlenecked by PCIe (current x8 Gen 1.0 at 1.6GB/s). We are currently upgrading BlueCache pipeline for PCIe Gen 2.0, which would double the bandwidth. In general, since the total throughput from multiple BlueCache servers is extremely high, a single appli-

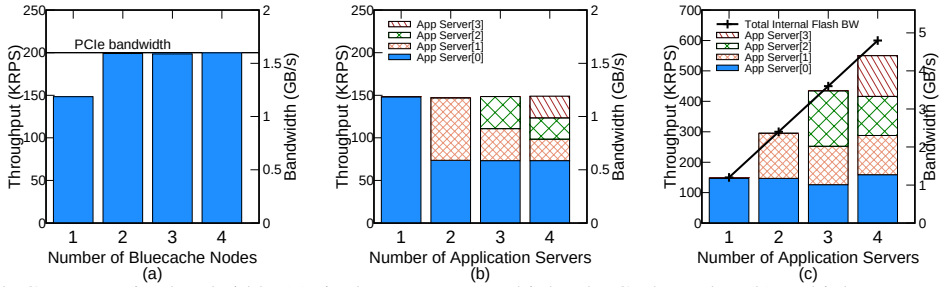


Figure 16: Multi-node GET operation bandwidth. (a) single app server, multiple BlueCache nodes, (b) multiple app servers, single Bluecache node, (c) multiple app servers, multiple BlueCache nodes

cation server with a host interface cannot consume the aggregate internal bandwidth of a BlueCache KVS cluster.

The second scenario examines the behavior of BlueCache when there is resource contention for a BlueCache storage node by multiple application servers. Figure 16(b) shows that the network engine is able to maintain the peak flash performance, but favors the application server to which it is attached. In the current implementation, half of the flash bandwidth is allocated to the application server attached to a storage node.

The last scenario illustrates the aggregated bandwidth scalability of BlueCache KVS cluster, with multiple application servers accessing all KVS nodes. The line in Figure 16(c) shows the total maximum internal flash bandwidth of all BlueCache nodes, and the stacked bars show overall throughput achieved by all application servers. We achieved 99.4% of the maximum potential scaling for 2 nodes, 97.7% for 3 nodes, and 92.7% for 4 nodes at total of 550.16 KRPS.

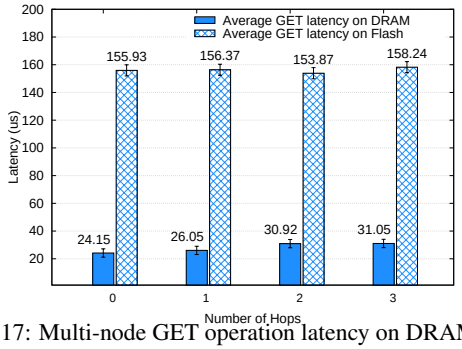


Figure 17: Multi-node GET operation latency on DRAM/Flash

2) Operation Latency: Figure 17 shows the average GET operation latency for 8KB key-value pairs over multiple hops of BlueCache nodes. Latency is measured from both DRAM and flash. We measured that that each hop takes ~0.5μs, therefore BlueCache handshake network protocol only takes 1.5μs per hop. When key-value pairs are on DRAM, we observed a ~2μs increase of access latency per hop for various number of node traversals, which is much smaller than overall access latency (~25μs). Access variations from other parts of BlueCache hardware are far greater than the network latency, as shown as error bars in Figure 17. Because accessing remote nodes are equally as fast as local nodes, the entire BlueCache KVS cluster appears as a fast local KVS storage, even though it is physically distributed among different devices.

6.3 Application Multi-Access Performance

Applications use BlueCache's software interface (See Section 4) to access BlueCache KVS cluster concurrently. Figure 18 shows the performance of BlueCache's software interface when there are multiple application threads sending *synchronous* GET requests. When there are a small number of application threads, BlueCache delivers nearly raw flash device latency of 200μs. However, flash bandwidth is not saturated because there are not enough outstanding requests. Increasing the number of concurrent threads increases in-flight KVS requests and KVS throughput. However, if there are too many threads, latency increases to the undesirable millisecond range because of significant software overheads from context switching, locking, and request clustering. Configuring the application with 128 threads delivers 132.552 KRPS throughput at only 640.90 μs average latency on 1KB key-value pair GETs.

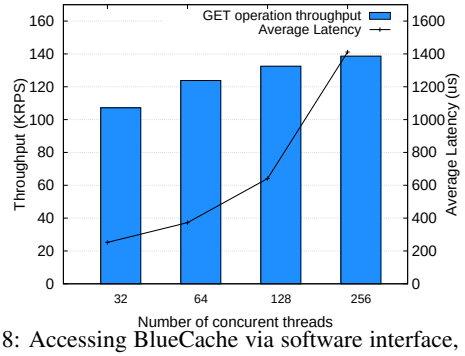


Figure 18: Accessing BlueCache via software interface, key-value pair sizes are 1KB

We also ran the same experiments on Fatcache, which is an SSD-backed software KVS implemention. We ran Fatcache on a 24-core Xeon server with one 0.5TB PCIe-based Samsung SSD and one 1Gbps Ethernet. The SSD performs like BlueCache's Flash board (130K IOPs vs 150K IOPs for 8KB page reads). Our result shows that Fatcache only provides 32KRPS throughput and 3000μs average latency. In comparison, BlueCache can fully exploit the raw flash device performance, and shows 4.18X higher throughput and 4.68X lower latency over Fatcache.

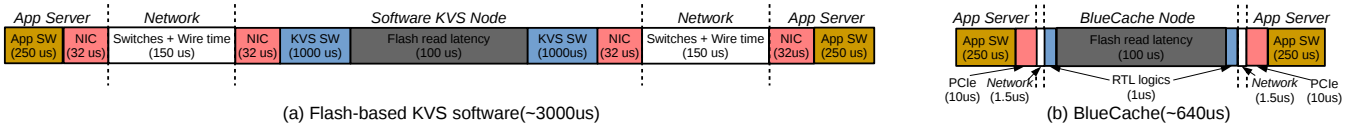
Figure 19 breaks down the latency of Fatcache and BlueCache requests. The actual Flash access latency is the same, but BlueCache eliminates 300μs of network latency [53], 128 μs of NIC latency [58, 53], and 2000 μs of KVS software latency.

6.4 Social Networking Benchmark

We used BG [13], a social networking benchmark, to evaluate BlueCache as a data-center cache. BG consists of a back-end storing profiles for a fixed number of *users*, and a front-end multi-threaded workload simulator. Each front-end thread simulates a user performing *social actions*. Social actions consist of *View Profile, List Friends, Post Comment, Delete Comment,* and so forth.

An example of a BG backend is a MySQL database. The MySQL database persistently stores member profiles in four tables with proper primary/foreign keys and secondary indexes to enable efficient database operations. BG implements 11 social actions which can be translated to SQL statements to access the database.

BG also supports augmenting the back-end with a KVS cache. A social action type is prefixed with the member ID to form a KVS request key, the corresponding MySQL query result is stored as the



(a) Flash-based KVS software(~3000us) (b) BlueCache(~640us)

Figure 19: End-to-end processing latency comparison of SSD KVS software and BlueCache

value. The interactions between KVS and MySQL database are the same as were described in Section 2.1. The average size of key-value pairs of BG benchmark is 1.54KB.

We performed experiments with BG benchmark by using three different KVS systems: BlueCache, a DRAM-based KVS (stock memcached) and a flash-based KVS (Fatcache [63]). In the next two sections we describe the experiment setup and three experiments to evaluate BlueCache.

6.4.1 Experiment setup

MySQL Server runs a MySQL database containing persistent user data of BG benchmark. It is a single dedicated machine which has two Intel Xeon E5-2665 CPUs (32 logical cores, 2.4GHz), 64GB DRAM, 3x 0.5TB M.2 PCIe SSDs in RAID-0 (~3GB/s bandwidth) and a 1Gbps Ethernet adapter. The database is pre-populated with member profiles of *20 millions users*, which is ~600GB of data. The database is configured with a 40GB InnoDB buffer pool.

Application Server runs BG's front-end workload simulator on a single machine which has two Intel Xeon X5670 CPUs (24 logic cores, 2.93GHz), and 48GB DRAM. BG's front-end multi-threaded workload simulator supports a maximum of 6 million active users on this server, which is about 20GB of working set. The front end has a zipfian distribution with mean value of 0.27, to mimic the skew nature of the workload.

Key-Value Store caches MySQL database query results. We experimented with three KVS systems to examine behaviors of different KVSs as data-center caching solutions.

System A, Software DRAM-based KVS: System A uses stock memcached as a in-memory key-value cache, and uses 48 application threads to maximize throughput of memcached.

System B, Hardware flash-based KVS: System B uses a single BlueCache node as a key-value cache. The BlueCache node is attached to the client server via PCIe. System B uses 128 application threads to maximize throughput of BlueCache.

System C, Software flash-based KVS: System C uses Fatcache, a software implementation of memcached on commodity SSD. System C uses 48 application threads to maximize throughput of Fatcache.

Table 4 compares the characteristics of storage medium used by three different KVS systems.

Table 4: KVS storage technology comparison

KVS systems	Storage Media	Capacity	Bandwidth
memcached	DDR3 SDRAM DIMMs	15GB	64GB/s
BlueCache	NAND flash chips	0.5TB	1.2GB/s or 150K IOPs
FatCache	Samsung m.2 PCIe SSD	0.5TB	1GB/s or 130K IOPs

We also found that running the benchmark frontend and memcached on the same server has higher throughput than running them on separate machines. The network connection speed between the application server and the KVS plays a critical role in the overall system performance. Figure 20 shows that the throughput of stock memcached decreases exponentially when more of the processing time is spent on network compared to actual in-memory KVS access. We measured peak throughput of stock memcached via 1Gbps Ethernet, which is about 113 KRPS and is lower than the peak throughput of BlueCache. The 1Gbps Ethernet can limit the performance of System A and C, while BlueCache's fast network is not a bottleneck for System B. When memcached and the application runs on the same server, the throughput is 423KRPS, and the server utilizes <50% of CPU cycles at peak, which means sharing CPU cycles was not a bottleneck in such a set-up. For our experiments, we eliminated the 1Gbps network bottleneck and deployed both System A and C on the same physical machine of the application server, to have a fair comparison with System B.

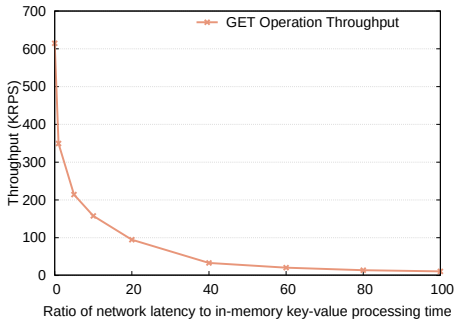


Figure 20: Throughput of memcached for relative network latencies

6.4.2 Experiments

We ran BG benchmark with our experimental setup to evaluate the characteristics of BlueCache and other KVSs as data-center caches. Misses from KVS are penalized by reading the slower backend to refill the KVS cache, thus KVS miss rate is an important metric to evaluate the efficacy of the cache. There are two types of misses for KVS caches. First type happens when KVS capacity is limited and can not hold the entire working set of the application. We call them *capacity misses*. The second type of KVS misses are caused by updates to the backend. When new data is written, the application DELETEs the relevant key-value pairs in KVS to make KVS coherent with the new data in the backend. In this case, the next corresponding GET request will return a miss. We call such misses *coherence misses*.

We ran three experiments to examine how KVS misses can effect the overall performance of different KVS cache solutions.

Experiment 1 evaluates the benefits of a slow and large KVS cache over a fast and small one. Like typical real use cases [18, 11], the front-end application in this experiment has a very low update rate (0.1%), thus coherence misses are rare. DRAM-based solution (memcached) is faster than flash-based solution (BlueCache), but its superiority can quickly diminish as the former suffers from more capacity misses. We will show the cross point where BlueCache overtakes memcached. We will also show the superiority of a hardware-accelerated flashed-based KVS solution over a software implementation (BlueCache vs. Fatcache).

Experiment 2 examines the behavior of BlueCache and memcached when they have the same capacity. In such cases, both KVS solutions will experience capacity misses, and we will show the performance drop as capacity misses increase for both systems.

Experiment 3 examines the behavior of BlueCache and memcached when the application has more updates to the backend. In such cases, both KVS solutions will experience coherence misses, and we will show the performance drop as coherence misses increase for both systems.

Results: *Experiment 1* shows that BlueCache can sustain more request rate than memcached, when the latter has the more than

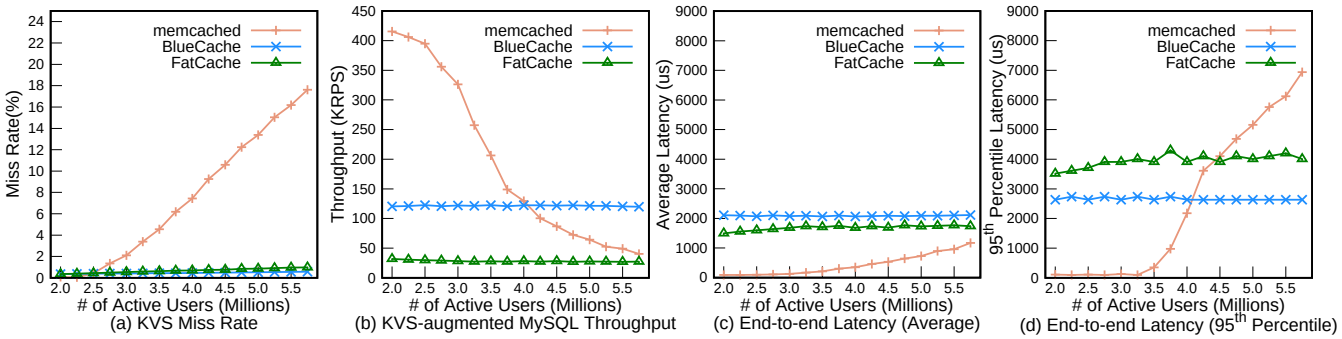


Figure 21: Performance of BlueCache and other KVS systems as augmentations to a MySQL database, which stores simulated data of a social network generated by BG benchmark. Evaluations are made with various numbers of active social network users.

7.4% misses (See Figure 21(a)(b)). With no capacity misses, memcached sustains 3.8X higher request rate (415KRPS) than BlueCache (123KRPS), and 12.9X higher than Fatcache (32KRPS). As the number of active users increases, the capacity of memcached is exceeded and its miss rate increases. Increasing miss rate degrades throughput of the overall system, since each memcached miss requires MySQL accesses. BlueCache and memcached meet at the cross point at 7.4% miss rate. Fatcache, on the other hand, is 4.18X slower than memcached, and its Fatcache throughput does not exceed that of memcached until the number of active users exceeds 5.75 million and memcached's miss rate exceeds 18%.

Similar trend can also be found in the end-to-end application latency of BG benchmark for different KVS solutions (Figure 21(c)(d)). When there are no capacity misses, memcached provides an average latency of 100μs that is 20X faster than BlueCache.When memcached suffers more than 10% misses, BlueCache shows much better 95th percentile latency (2600μs) than memcached (3600μs-6900μs), because of miss penalties by MySQL. On the other hand, Fatcache shows about similar average latency with BlueCache (1700μs vs. 2100μs), but it has much larger variations in the latency profile and has 1.5X shorter 95th percentile latency than BlueCache (4000μs vs. 2600μs).

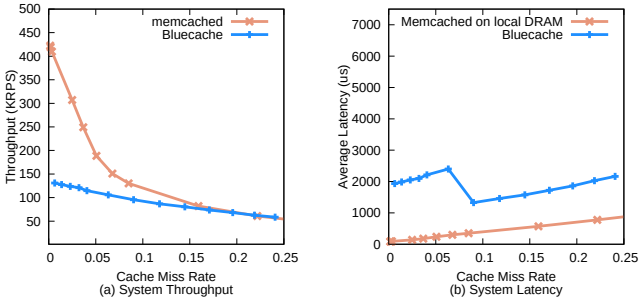


Figure 22: BG performance for different capacity miss rates, *both memcached and BlueCache have the same capacity of 15GB*

Experiment 2 examines the behavior of capacity cache misses of BlueCache and memcached. In this experiment, we configured the BlueCache to the same size of as memcached of 15GB, and made the benchmark to issue *read-only* requests. Thus, all misses are capacity misses. Figure 22(a) shows that throughput of both KVSs decreases as miss rate increases, with memcached dropping at a faster pace. Beyond 16% miss rate, both KVSs merge at the same throughput when the backend becomes the bottleneck. Figure 22(b) shows that latency also increases as miss rate increases, but memcached delivers shorter latency than BlueCache in general. On Figure 22(b), there is a breakpoint of the latency for BlueCache, because of change of thread counts of the benchmark. The benchmark changes the thread count from 128 to 64 at the breakpoint, in order to reduce request congestion and lower latency while maintaining maximum throughput. In an ideal case, the benchmark would dynamically adjust the request admission rate, so that the curve in Figure 22(b) would be smooth.

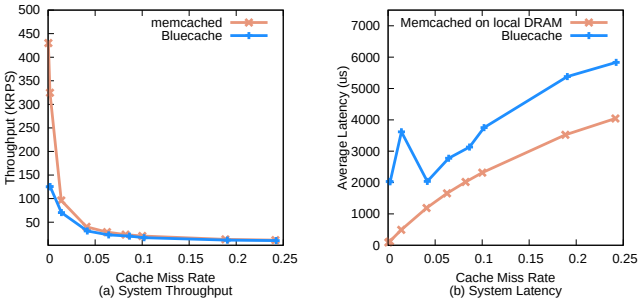


Figure 23: BG performance for different coherence miss rates

Experiment 3 examines the behavior of coherence cache misses of BlueCache and memcached. In this experiment, we vary read-/write ratios of the benchmark, to control the coherence cache miss rate. Similar to Experiment 1, the throughput of both KVSs decreases as miss rate increases, with memcached dropping at a faster pace (Figure 23(a)). And at merely 6% miss rate, both KVSs merge at the same throughput when the backend becomes the bottleneck. Figure 23(b) shows that latency also increases as miss rate increases, but memcached delivers shorter latency than BlueCache in general. Similar to Experiment 2, the break point of latency for BlueCache is due to change of request thread count to reduce request congestion from the applications.

7. CONCLUSION

We have presented BlueCache, a fast distributed flash-based key value store appliance that uses near-storage KVS-aware flash management and integrated network as an alternative to the DRAM-based software solutions at much lower cost and power. A rack-sized BlueCache mini-cluster is likely to be an order of magnitude cheaper than an in-memory KVS cloud with enough DRAM to accommodate 10~20TB of data. We have demonstrated the performance benefits of BlueCache over other flash-based key value store software without KVS-aware flash management. We have demonstrated the scalability of BlueCache by using the fast integrated network. Moreover, we have shown that the performance of a system which relies data being resident in in-memory KVS, drops rapidly even if a small portion of data has to be stored in the secondary storage. With more that 7.4% misses from in-memory KVS, BlueCache is superior solution in data centers than a DRAM-based KVS, with more affordable and much larger storage capacity.

All of the source codes of the work are available to the public under the MIT license. Please refer to the git repositories: `https://github.com/xushuotao/bluecache.git` and `https://github.com/sangwoojun/bluedbm.git`.

Acknowledgments. This work is partially funded by Samsung (Res. Agmt. Eff. 01/01/12), SK Hynix, Netapp and MIT Lincoln Laboratory (PO7000261350) under various grants to CSAIL, MIT. We thank Quanta Computer for providing help and funding (Agmt. Dtd. 04/01/05) in building BlueDBM on which all our experiments were performed. We also thank Xilinx for their generous donation of VC707 FPGA boards and FPGA design expertise.

8. REFERENCES

[1] A Hash Function for Hash Table Lookup. http://goo.gl/VDzzLb.
[2] Bluespec Inc. http://www.bluespec.com.
[3] Netflix EVCache. http://goo.gl/9zoxJ6.
[4] Nvidia GeForce GTX 780 Specifications. http://goo.gl/6Yhlv6.
[5] Redis. http://redis.io.
[6] Samsung 850 PRO. http://goo.gl/vjPj7V.
[7] Samsung 950 PRO. http://goo.gl/DCwQpd.
[8] Samsung M393A2G40DB0-CPB. http://goo.gl/BOL4ye.
[9] Samsung DDR4 SDRAM. http://goo.gl/LO1ExG, June 2013.
[10] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, pages 1–14, 2009.
[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, pages 53–64, 2012.
[12] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *NSDI*, pages 211–224, 2011.
[13] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.
[14] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don'T Thrash: How to Cache Your Hash on Flash. *Proc. VLDB Endow.*, pages 1627–1637, 2012.
[15] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core Key-value Store. In *IGCC*, pages 1–8, 2011.
[16] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.
[17] M. Blott, L. Liu, K. Karras, and K. Vissers. Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory. In *HotStorage*, 2015.
[18] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC*, pages 49–60, 2013.
[19] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA Memcached Appliance. In *FPGA*, pages 245–254, 2013.
[20] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA*, pages 266–277, 2011.
[21] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *EUC*, pages 394–404, 2006.
[22] B. K. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.*, pages 1414–1425, 2010.
[23] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *NSDI*, pages 401–414, 2014.
[24] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadahisa, T. Asai, and M. Motomura. Caching memcached at reconfigurable network interface. In *FPL*, pages 1–6, 2014.
[25] Fusion IO. using membrain as a flash-based cache. http://goo.gl/Khecz6, December 2011.
[26] S. Gunelius. The Data Explosion in 2014 Minute by Minute Infographic. http://goo.gl/9CqKj5, July 2014.
[27] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *SoCC*, pages 43–57, 2015.
[28] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. In *ISPASS*, pages 88–98, 2012.
[29] Intel Inc. Intel Data Direct I/O Technology. http://goo.gl/2puCwN.
[30] Intel Inc. Intel Data Plane Development Kit(Intel DPDK) Overview - Packet Processing on Intel Architecture. http://goo.gl/W5oBBV, December 2012.
[31] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *CCGRID*, pages 236–243, 2012.
[32] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *ICPP*, pages 743–752, 2011.
[33] S.-W. Jun, M. Liu, K. E. Fleming, and Arvind. Scalable Multi-access Flash Store for Big Data Analytics. In *FPGA*, pages 55–64, 2014.
[34] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. BlueDBM: An Appliance for Big Data Analytics. In *ISCA*, pages 1–13, 2015.
[35] S.-W. Jun, M. Liu, S. Xu, and Arvind. A transport-layer network for distributed fpga platforms. In *FPL*, pages 1–4, 2015.
[36] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *SIGCOMM*, pages 295–306, 2014.
[37] D. Karger, E. Lehman, T. Leighton, R. Panigraphy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, pages 654–663, 1997.
[38] M. King, J. Hicks, and J. Ankcorn. Software-Driven Hardware Development. In *FPGA*, pages 13–22, 2015.
[39] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *ICCSA*, pages 151–156, 2008.
[40] M. Lavasani, H. Angepat, and D. Chiou. An FPGA-based In-Line Accelerator for Memcached. *Computer Architecture Letters*, pages 57–60, 2014.
[41] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *FAST*, pages 273–286, 2015.
[42] S. Lee, J. Kim, and Arvind. Refactored Design of I/O Architecture for Flash Storage. *Computer Architecture Letters*, pages 70–74, 2015.
[43] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-Managed Flash. In *FAST*, pages 339–353, 2016.
[44] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD*, pages 1075–1086, 2008.
[45] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *ISCA*, pages 476–488, 2015.
[46] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *SOSP*, pages 1–13, 2011.
[47] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *NSDI*, pages 429–444, 2014.
[48] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *ISCA*, pages 36–47, 2013.
[49] M. Liu, S.-W. Jun, S. Lee, J. Hicks, and Arvind. minflash: A minimalistic clustered flash array. In *DATE*, pages 1255–1260, 2016.
[50] X. Liu and K. Salem. Hybrid Storage Management for Database Systems. *Proc. VLDB Endow.*, pages 541–552, 2013.
[51] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, pages 103–114, 2013.
[52] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, pages 385–398, 2013.
[53] J. Ousterhout. RAMCloud and the Low-Latency Datacenter. http://goo.gl/uWsPnu, 2014.
[54] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, pages 92–105, 2010.
[55] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *ASPLOS*, pages 471–484, 2014.
[56] X. Ouyang, N. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. Panda. SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks. In *ICPP*, pages 470–479, 2012.
[57] I. Petrov, G. Almeida, A. Buchmann, and U. Gräf. Building Large Storage Based On Flash Disks. In *ADMS@ VLDB*, 2010.
[58] M. Rosenblum and A. N. Mario Flajslik. Low Latency RPC in RAMCloud. http://goo.gl/3FwCnU, 2011.
[59] SanDisk. Fusion ioMemory PX600 PCIe Application Accelerators. http://goo.gl/rqePxN.
[60] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *USENIX ATC*, pages 14–14, 2010.
[61] R. Stoica and A. Ailamaki. Improving Flash Write Performance by Using Update Frequency. *Proc. VLDB Endow.*, pages 733–744, 2013.
[62] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *USENIX ATC*, pages 347–353, 2012.
[63] Twitter Inc. Fatcache: memcache on SSD. https://github.com/twitter/fatcache.
[64] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. VLDB Endow.*, pages 1226–1237, 2015.