High-Performance Key-Value Store On OpenSHMEM

Huansong Fu* Manjunath Gorentla Venkata† Ahana Roy Choudhury* Neena Imam† Weikuan Yu*
Florida State University* Oak Ridge National Laboratory†
{fu,roychou,yuw}@cs.fsu.edu {manjugv,imamn}@ornl.gov

***Abstract*—Recently, there has been a growing interest in enabling fast data analytics by leveraging system capabilities from large-scale high-performance computing (HPC) systems. OpenSHMEM is a popular run-time system on HPC systems that has been used for large-scale compute-intensive scientific applications. In this paper, we propose to leverage OpenSHMEM to design a distributed in-memory key-value store for fast data analytics. Accordingly, we have developed SHMEMCache on top of OpenSHMEM to leverage its symmetric global memory, efficient one-sided communication operations and general portability. We have also evaluated SHMEMCache through extensive experimental studies. Our results show that SHMEMCache has accomplished significant performance improvements over hte original Memcached in terms of latency and throughput. Our initial scalability test on the Titan supercomputer has also demonstrated that SHMEMCache can scale to 1024 nodes.**

I. Introduction

The computation power and storage capacity of High-Performance Computing (HPC) systems have been advancing rapidly for several decades. While these capabilities have been used to solve computation-intensive scientific problems, there has been a rise in the interest to leverage HPC systems for big data analytics. Particularly, the U.S. National Strategic Computing Initiative has mandated large-scale data analytics as a primary objective for upcoming exascale systems. Several prior studies have attempted to use HPC mechanisms in order to develop various components for high-speed distributed data analytics. For example, RDMA, which is a popular communication technique on HPC systems, has been utilized to build a variety of key-value (KV) stores [18], [23], [11], [29], [27]. In the meantime, PGAS (Partitioned Global Address Space) programming models have gained wide popularity on many HPC systems. As one of the PGAS models, Symmetric Hierarchical Memory access (SHMEM) has been widely used for more than two decades to write and execute parallel programs. OpenSHMEM [9], which is a standardization effort of SHMEM, has drawn increasing attention from both industry and academia [26], [16], [15].

In light of its increasing popularity, we have examined the potential of OpenSHMEM in supporting data analytics, and found that it offers a number of great opportunities for developing a KV store. First, OpenSHMEM has similar memory-style addressing that makes it a good match for building in-memory KV stores. The visibility of OpenSHMEM's *symmetric memory* can help clients to directly access KV pairs. In addition, OpenSHMEM's *one-sided communication* can relieve the server of participation in communication and improve the KV store's communication efficiency. Furthermore, OpenSHMEM can deliver good performance on a variety of platforms [25]. Such portability can facilitate the deployment of distributed KV stores on both commodity servers and the HPC systems without significant increase in the amount of effort required to do so.

In this paper, we describe the design of *SHMEMCache* as an in-memory KV store on top of the salient mechanisms from OpenSHMEM including its global symmetric memory and one-sided communication operations. Compared to our initial work described in [13], we have further introduced several novel techniques to SHMEMCache in order to deliver low latency, high throughput, and good scalability, while ensuring data consistency.

While SHMEMCache can leverage OpenSHMEM's symmetric memory and support remote read or write operations from the clients through one-sided operations, conflicts can happen when multiple concurrent operations are issued to the same KV pair. These are often categorized as *read-write* and *write-write races* [23], [29]. To resolve these two races, we have introduced a mechanism called *read-friendly exclusive write*. It allows concurrent reads to verify the data consistency through a lightweight version check while precluding concurrent writes using locks.

In addition, traditional KV store such as Memcached [2] uses linked lists to resolve hash collisions, causing the problem of *pointer chasing*, where it needs to follow multiple pointers to locate a KV pair. This problem becomes costly when the client tries to locate a KV pair remotely in SHMEMCache, which we refer to as *remote pointer chasing*. To mitigate its impact, we have designed a *set-associative hash table* that accommodates multiple pointers in one hash entry and a per-client *pointer directory* that caches remote pointers for fast access.

Furthermore, managing cached KV pairs in SHMEMCache is also challenging. In SHMEMCache, a server may not be aware of clients accessing KV pairs while it is deciding whether to evict KV pairs based on the least recently used (LRU) policy. On the other hand, the client is unaware of the server's decision to evict KV pairs, so it cannot invalidate its cached pointers for those KV pairs. Explicit synchronization about the access or eviction history between the server and the client could incur significant network and processing overheads. To solve those issues, we relax the definition of

the *recency* of a KV pair from an exact time point of its last access to a small *time range* that its last access belongs to. This relaxation enables us to update the recency of KV pairs and inform client about evicted KV pairs in a coarse-grained manner, without incurring excessive overheads.

We have implemented these aforementioned techniques in SHMEMCache which we have developed as a distributed in-memory KV store on top of OpenSHMEM. We have also evaluated SHMEMCache's performance using both microbenchmarks and the YCSB benchmark [10]. The results show that SHMEMCache not only has more than an order of magnitude of performance improvement compared to Memcached in terms of operation latency and throughput, but also outperforms the KV store in HiBD [1] that was developed on top of InfiniBand RDMA [18] by the Ohio State University. In addition, we have evaluated SHMEMCache on the Titan supercomputer at Oak Ridge National Laboratory. Our initial results from a small-scale evaluation have shown that SHMEMCache can achieve good scalability for execution on up to 1024 compute nodes.

II. Background and Challenges

In this section, we first briefly introduce Memcached and OpenSHMEM, and then discuss the challenges encountered while designing SHMEMCache as a KV store on top of OpenSHMEM.

A. Background

1) Memcached: The demand to provide low-latency and high-throughput access to persistent databases has fostered a significant amount of interest in distributed in-memory KV stores such as Memcached. For example, Facebook [24] and Twitter [6] both use Memcached to meet their massive caching requirement. As shown on the left side of Fig. 1, Memcached has a *client-server* architecture where the client can communicate with the server to perform KV operations. Two main KV operations are `SET`, i.e. inserting a new KV pair or updating an existing KV pair, and `GET`, i.e. retrieving the value associated with a given key. The server allocates many memory slabs that are divided into different sizes of small blocks. For each slab, it manages a *free list* that contains pointers to all unused blocks in the slab. Each KV pair is stored in the *smallest possible* KV block. The lack of available blocks when inserting new pairs will trigger *cache eviction*, for which Memcached uses the LRU algorithm to evict the *cold* pairs, namely, pairs that have the least recent access time. Memcached uses consistent hashing to locate a given KV pair. If multiple pairs are being hashed to the same hash table entry, only the most recently accessed pair will appear in the entry and the others will be chained after it.

2) OpenSHMEM: OpenSHMEM is a PGAS library interface specification for standardizing the SHMEM library interfaces. Process in OpenSHMEM is called *Processing Element* (PE). Each PE has a local memory space and a symmetric memory space. The symmetric memory contains symmetric variables, which exist with the same name, type and relative address at all the PEs. OpenSHMEM uses one-sided communication operations to transfer data from one PE to the symmetric memory of other PEs. Two representative one-sided operations in OpenSHMEM are `put`, i.e., copying local data to remote symmetric memory, and `get`, i.e., copying data from remote symmetric memory to local memory. In this paper, we use `shm_put` and `shm_get` to denote them. This is done in order to avoid confusion with the `SET` and `GET` KV operations.

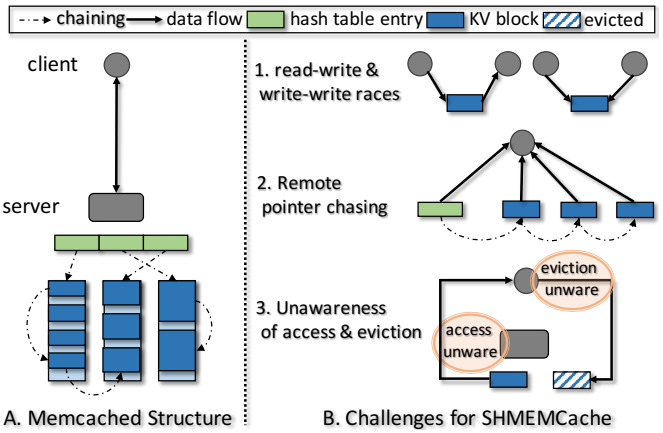


Fig. 1: Memcached and challenges for SHMEMCache.

B. Challenges for Building a KV Store on OpenSHMEM

We have analyzed the internal architecture of Memcached and the memory and communication features of OpenSHMEM, and accordingly, have identified three major challenges for SHMEMCache. In the remaining part of this section, we will discuss the challenges in detail.

1) Supporting KV Operations with One-sided Communication: Leveraging OpenSHMEM's one-sided `shm_get` and `shm_put` to support the `GET` and `SET` KV operations requires an in-depth examination of their compatibility. Allowing remote clients to directly read or write KV pairs can result in severe data inconsistency as well as some other issues. On one hand, using one-sided `shm_get` for `GET` leads to read-write races as shown in Fig. 1. A reader may read a corrupted KV pair because a writer is modifying the pair at the same time. Most of the related works adopt the optimistic concurrency control that allows multiple readers to read the same KV pair without locking. In those works, the solutions to the read-write races include asking clients to compute the checksum of the KV pair [23] or comparing versions of every cache line that form the KV pair [11]. However, the former solution introduces heavy computation and the latter one needs to strip off the cacheline versions and copy the KV pair content out before processing the pair. These costs grow linearly with the size of the KV pair. Moreover, both costs are added to every `GET`, which could make the solutions particularly costly to the read-dominant workloads in real-world scenarios [8]. Another feasible solution is out-of-place update [29]. However, because it applies update in a new block, the client that reads the old block will need to look up the pair again.

On the other hand, using one-sided `shm_put` for `SET` has a number of additional difficulties. The first is the difficulty for the client to perform remote management of data structures in the server. For example, inserting a new pair will require the client to request an unused block in the free list, to delete the block from the free list and to modify an entry in the hash table. In addition, write-write races can occur and multiple writes to the same KV pair may jeopardize its consistency. Finally, the client needs additional confirmation from the server that it is working on the targeted KV pair. Due to these difficulties, most of the related works only allow the local server to handle all `SET`s. However, this approach could not take full advantage of the benefits of one-sided communication operations from OpenSHMEM, being unable to fully tap the server's processing power for maximal system throughput.

2) Remote pointer chasing: In a KV store, pointer chasing means that a KV pair lookup in the hash table may not result in fetching the desired pair. Instead, the server may traverse through a long list of pairs before it gets to the desired one. This issue becomes more serious when the client can remotely access the hash table and KV store, causing remote pointer chasing (Fig. 1). This brings more performance concerns because multiple network roundtrips will take much longer than multiple local memory accesses, even with one-sided communications [11].

Related works mainly involve attempts to design more efficient hash functions or hash table designs, including Cuckoo hashing [12], [23], Hopscotch hashing [11], and cache-friendly hash table [29]. However, there has been little consensus on which hash function or table design is superior over others. Another practical solution is to store pointers on the client side so the client can use them to directly access the KV pairs without remote pointer chasing [28]. However, little detail has been revealed on how the cached pointers are organized.

3) Cache Management: As SHMEMCache leverages the one-sided communication operations, there are two issues associated with the cache management of KV pairs: the server is unaware of the client remotely accessing KV pairs and the client is unaware of the server evicting pairs, both of which will bring challenges to SHMEMCache.

Firstly, most of the existing KV stores use the least recently used (LRU) policy or a LRU-approximation for cache eviction. In a traditional KV store such as Memcached, implementing the LRU cache eviction does not involve many complexities. However, it becomes a problem when the server is unaware of the client accessing the KV pairs. Letting the client notify the server after every remote access is not practical as it could negate the benefit of one-sided operations. A previous work [28] has used a solution that asks the client to record its recent access history periodically and send it to the server. However, preparing and processing such history report will interfere with normal processing and add significant overheads on both sides.

Secondly, if the client temporarily caches pointers to mitigate remote pointer chasing, it needs to know when to invalidate cached pointers for the KV pairs that have been evicted by the server. Otherwise, it may access a wrong pair and this will increase the average operation latency. Also, the space for pointer caching could get filled with a lot of invalid pointers that makes it inefficient. Similar to the previous issue, it is impractical for the server to faithfully notify the client after the eviction of every KV pair due to performance concerns. A previous work has proposed a *lease*-based mechanism to solve this issue [28]. The basic idea is to use a lease as an agreement between the server and the client on whether the pointer of a KV pair is valid at a given point. However, the mechanism entails several complicated problems. For example, it is difficult to assign an optimal lease for each pair with little prior knowledge about the pair. Moreover, an expired lease does not essentially indicate eviction of the pair, which leads to a premature invalidation of the pair's pointer in the client.

III. Design of SHMEMCache

The design of SHMEMCache strives to tackle the challenges identified in Section II-B and provide (1) high-performance KV operations with date consistency, (2) avoidance of remote pointer chasing and (3) efficient cache management. In this section, we will first give an overview of the design. Then, we will elaborate on each part mentioned above.

A. Overview

Fig. 2 illustrates the main structure of SHMEMCache. In SHMEMCache, an OpenSHMEM PE can have the role of either a client or a server. A server uses the symmetric memory space from OpenSHMEM to host three structures: the hash table, the KV store and the *message chunks*. The hash table has a *set-associative* structure to accommodate multiple pointers in one table entry. A lookup operation to the hash table fetches multiple pointers from the matching entry. The KV store consists of many blocks to store individual KV pairs. Similar to Memcached, SHMEMCache categorizes the blocks by sizes and stores a KV pair in the smallest possible block. The client conducts **Direct KV Operations** on the KV pairs, which directly write or read the KV pairs without server's involvement, i.e. Direct `SET` or Direct `GET`. The message chunks are fixed-size memory spaces, which the client can directly write to. The client conducts **Active KV Operations** by writing messages that incorporate information of KV operations to the message chunks and asking the server to process those KV operations. The use of Active KV Operations happens when there is no available pointer to the target KV pair for conducting Direct KV operations.

A client uses a small symmetric memory space for its own message chunks so that the server can also write messages to them. In addition, the client features a local *pointer directory* to cache pointers of KV pairs that it has recently accessed.

We enable the client to use OpenSHMEM's one-sided `shm_put` and `shm_get` to conduct the Direct `SET` and `GET`. To ensure data consistency, we introduce the *read-friendly exclusive write* to resolve the read-write and write-write races. Specifically, we address the read-write races by adding two

additional small writes before and after the exclusive write for a KV pair. The reader only needs to check the version number of the two small writes to verify the integrity of the KV pair. Since this approach only incurs negligible overheads for `GET`, it is more friendly towards read-dominant workloads. As for write-write races, we use OpenSHMEM's atomic operations to implement a locking mechanism. Before an exclusive write to a KV pair, we claim the lock of the corresponding KV block. However, not all the KV operations can be conducted directly due to the lack of corresponding pointers in the hash table or pointer directory. In this case, the client will conduct the Active KV operations.

To tackle the issue of remote pointer chasing, SHMEMCache provides a set-associative hash table, in which each entry can accommodate multiple pointers. However, this cannot eliminate the issue because the space in an entry is still limited. Making the entry too large results in long latency of remote hash table lookup. Thus, each client also has a local pointer directory to cache pointers of both the most recently and frequently accessed KV pairs. This pointer directory has similar organization as the set-associative hash table, allowing fast retrieval of pointers and efficient using of memory.

Finally, we enable a coarse-grained mechanism for both recency update and cache eviction to address the high cost in updating the corresponding information. We use a relaxed *time range* instead of an exact time to indicate the recency of KV pairs. In order to update the recency without server processing, the client leverages the one-sided atomic operation to conduct the update remotely. In order to minimize network overheads, the update happens only when the recency of the key-value pair is changed to a different time range. Then, to ensure that the client gets informed about the evicted KV pairs, we first let the server maintain a number of linked lists called *recency tiers*. Each recency tier contains pointers of the KV pairs that have the same recency. When the need for cache eviction arises, the server will conduct a *batch eviction* that evicts all the KV pairs in the oldest recency tier at once. After that, it informs the client about the time range of evicted pairs through a small message and the client will *lazily invalidate* the pointers of evicted pairs based on the received time range.

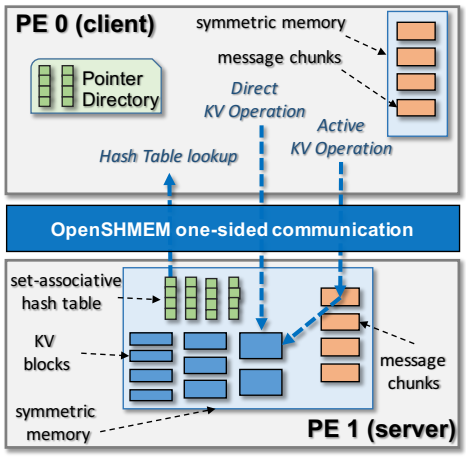


Fig. 2: Overview of SHMEMCache.

Note that, only one client/server is shown on a PE in the structure in Fig. 2. We have provided a communication mechanism using shared memory between the PE and the client/server so that a PE can host multiple clients/servers on-the-fly. More details can be found from our previous work on using the PEs from OpenSHMEM as communication delegators to enable Memcached [13].

B. KV Operations in SHMEMCache

A client can perform either Direct or Active KV operation upon a `SET`/`GET`. As mentioned before, remote management of data structures in the server can be difficult when the client is trying to insert a new KV pair. This observation leads us to our design choice to only leverage Direct `SET` for updating an existing pair. Then, we have a decision process for the client to determine if it needs to use a Direct or an Active KV operation. First, if the pointer of given KV pair can be found in the local pointer directory, the client will conduct Direct KV operations. Otherwise, we treat `SET` and `GET` differently. For a `SET`, the client performs the Active `SET`. For a `GET`, the client first fetches the corresponding hash table entry for the pair. When the pointer can be retrieved from the hash table, the client conducts a Direct `GET`, or an Active `GET` if not. We do not conduct remote hash table lookup for `SET` mainly because the lookup for new-inserting pairs will incur unnecessary network roundtrips. This decision process ensures that only the operations for new-inserting pairs and operations on cold pairs need to be conducted as the Active KV operations which need server processing. These only account for a small portion of the real-world workloads [8].

Direct KV operations introduce write-write and read-write races, as discussed in Section II-B. Active KV operations require to have a messaging mechanism on top of OpenSHMEM, through which the client can send operation requests to the server. Next, we first describe a read-friendly exclusive write mechanism to solve the two races, and then an efficient messaging conduit that leverages OpenSHMEM's global address space and one-sided operations.

1) Read-friendly Exclusive Write: To resolve write-write races and the need of verifying the targeted pair, we leverage one-sided *atomic compare-and-swap* operation (CAS in short) to implement a locking and verification mechanism. As shown in Fig. 3, we place a *target word* at the end of the KV block as the comparing target for the CAS. The target word consists of a *tail version*, a *tag* and a *lock bit*. The tag is a short hash identifying the KV pair which differs from its hash value for the hash table. The lock bit indicates either a *locked* or an *unlocked* status. The tail version is an incremental version number which is only updated by a `SET`.

Note that, both writer and reader can be either a local server or a remote client. Before the exclusive write, the writer (*Writer A* in Fig. 3) claims the lock by conducting a CAS on the last word in the block. If all the tag, version number and lock bit match with the CAS condition given by the writer, the CAS will swap the lock bit with a locked status. The writer then carries out the exclusive write, including the KV pair,

the target word and some unused space. But the unused space is small since the KV pair is stored in the smallest possible block. This write will also flip the lock bit to the unlocked state. However, when the target word fails to match, the writer has different actions according to the return value: if only the lock bit does not match, the writer waits for a certain back-off and tries the lock again (*Writer B*); if the lock bit matches but the tag does not match, the writer will instead issue an Active `SET` to the server without any back-off; finally, if both the lock bit and tag match but the version number fails to match, the writer will retrieve the version number from the return value and use it to conduct another CAS without any back-off.

Then, in order to resolve read-write races, the writer needs an additional CAS to swap the head version to the same number as the tail version. We use CAS instead of normal writes to avoid conflicts with a former or subsequent write, in some rare cases. The reader, on the other hand, simply reads the whole KV block and only if the head version equals the tail version, the read succeeds (second read of *Reader A* in Fig. 3). Otherwise, the reader will try again after a certain back-off (first read of *Reader A* and *B*). Note that, this mechanism requires correct message ordering of the writes. For systems that do not support the in-order semantics, we use OpenSHMEM's `fence` call in between those writes. This will ensure the completion of individual writes and their ordering. The overheads of this mechanism are mainly due to two additional small CAS, and are, hence, negligible to the reader.

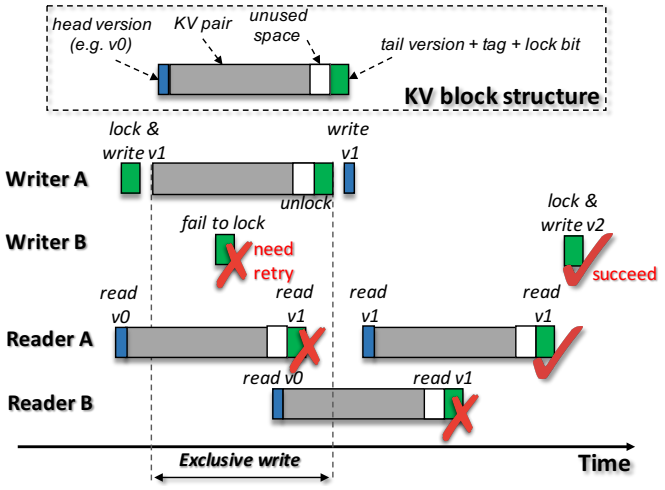


Fig. 3: Illustration of the exclusive write.

2) Messaging Conduit on OpenSHMEM: We have designed a messaging conduit on top of OpenSHMEM to serve the Active KV operations and other receiver-aware messaging. As shown in Fig. 4, on every PE, there is a doubly linked list of message chunks for every other PE. A sender PE can send messages to the corresponding message chunks on a receiver PE using one-sided `shm_put`. While polling, the receiver first polls a *use_flag* that indicates the arrival of the message. If a message is available, the receiver then polls a *comp_flag* at the end of the message, which indicates the completion of the data transfer. After that, it reads the message and processes it accordingly. Finally, it zeroes out the chunk. The message can be larger than the chunk, in which case, we divide a large message into multiple small messages and send them sequentially. Moreover, to better saturate the network, a client can send multiple messages at once, which is achievable because all the chunks in the symmetric memory are continuous.

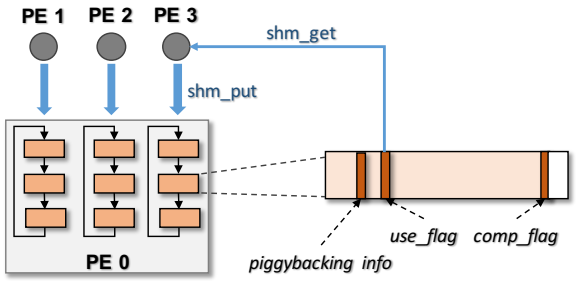


Fig. 4: Illustration of the messaging conduit.

However, the sender needs to know if the receiver has processed the data it has sent in order to mark those message chunks as available. We have two mechanisms that work together to get the sender informed. Firstly, similar to [22], within every message that is sent in the opposite direction, i.e., from the aforementioned receiver to the sender, the message piggybacks the corresponding information. Secondly, when there is no such piggyback opportunity, e.g., when client does not ask for acknowledgment from server, the sender will check the chunk itself. This is accomplished by leveraging the one-sided `shm_get`. When the sender finds that there are no available chunks in the receiver, it will fetch the `use_flag` from a chunk that is a few positions away from the last known available chunk. If the chunk being checked is available, the sender will mark all chunks between those two chunks as available.

C. Hash Table and Pointer Caching

Next, we describe how we use a set-associative hash table and a pointer directory structure to mitigate the impact of remote pointer chasing.

1) Set-Associative Hash Table: In the set-associative hash table, each entry consists of multiple sub-entries for multiple KV pairs. A sub-entry contains the pointer and tag of a KV pair. A lookup in the hash table will return all sub-entries in the corresponding table entry. Comparing the tag of the requested KV pair with the tags in the entry will help locate the KV pair. Note that there can be more pairs being hashed to the same table entry. In that case, the pointers that overflow will still be linked with one another after the last sub-entry. As discussed in Section III-B, the client will not chase the pointers that overflow but will issue Active KV operations instead.

2) Pointer Directory: Similar to the set-associative hash table, we organize the pointer directory with many entries, each comprise of several sub-entries. Every sub-entry in the pointer directory contains the tag, pointer, *access count* and *recency* of a KV pair. When searching for pointer of a pair, the client first locates the entry using the pair's main hash value, i.e., the one used for the hash table. Then, it tries to find the matched tag in the sub-entries.

Every successful Direct KV operation results in an update in the pointer directory. Also, if the client requests an acknowledgment message from the server for the Active KV operation, the acknowledgment message will contain the pointer of the KV pair and it also results in an update in the pointer directory. An update first checks if the pointer exists in the directory. If yes, the corresponding access count is incremented by 1 and the recency will be updated to the current time range. If the pointer does not exist, an empty sub-entry will be used. If there is a lack of free space, an existing pointer will be kicked out based primarily on the least recency and secondarily on the least access count.

D. Cache Management

Due to the high cost in explicitly exchanging information about every Direct KV operations or evicted pair, we take a coarse-grained approach to address the two challenges for cache management of SHMEMCache mentioned in Section II-B. Instead of rigorously defining the recency as the exact time of the most recent access, we relax this definition as a time range that the most recent access falls into. Each time range is defined with a *timestamp* and a *range*. For example, a time range with a timestamp of 1000 ms and a range of 100 ms indicates the time duration between 1000 to 1100 ms. The range is predetermined, so we only need the timestamp to denote a time range and also the recency. Such relaxation facilitates efficient cache management without hindering the normal eviction process as long as the range is reasonably small. Next, we first describe a *non-intrusive* approach to update the recency without disrupting server processing. Then we introduce a *batch eviction* mechanism on the server to evict cold pairs and a *lazy invalidation* mechanism on the client to invalidate expired pointers, both in a coarse-grained manner.

1) Non-intrusive Recency Update: As mentioned previously, the client will update its pointer directory when it has accessed a KV pair. Meanwhile, an update will also be applied to the server-side KV pair that updates the recency of the pair. However, aggressively updating the server-side recency upon every access can incur substantial data transfer overhead between client and server. Thus, the client only updates the recency when the recency changes. Since we have relaxed recency from a time point to a time range, all the other accesses of that pair within the same recency will not incur the recency update until their access times fall into a different recency. This avoids frequent data transfers between clients and servers. In addition, to avoid disrupting server processing for the update, the client conducts a CAS to the corresponding recency in the server-side KV store. Within the CAS, it provides the old recency and compares it with the target recency. Only a successful comparison of the recency value will result in updating the recency. Otherwise, the client does not apply the update. Meanwhile, whenever a pair is accessed through the Active KV operations, its recency is piggybacked within the sending message. The server will update it after conducting the KV operation. The server uses the recency generated by the client but does not generate it by itself in order to keep the client's pointer directory and its own KV store on the same page.

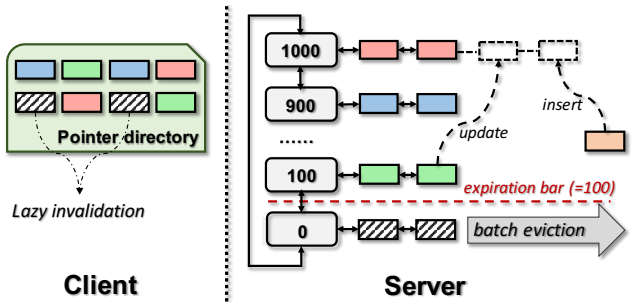


Fig. 5: Batch eviction and lazy invalidation. Each colored box represents a pointer to some KV pair with certain recency.

2) Batch Eviction and Lazy Invalidation: Notifying each client upon an eviction of a KV pair can lead to a flood of small messages. Thus, we let the server evict pairs in a batch, which is defined as all the pairs with the same recency. As shown in Fig. 5, on the server-side, we organize a number of recency tiers in a doubly linked list, and each will link many pointers of KV pairs, which are also organized in a doubly linked list. Each tier has its own recency. The top tier has the most recent recency and the bottom tier has the least recency among all the current tiers. The pairs that belong to the same tier have the same recency with that tier. When a new KV pair is inserted to KV store, its pointer will be inserted to the top recency tier as well. When an existing KV pair is updated in the KV store, its pointer is removed from its original tier and inserted to the current top tier. The top tier is changed regularly. When the current time exceeds the time range of the current top tier, an empty tier will be inserted as the new top tier. When the server needs to evict KV pairs, it will conduct a batch eviction that evicts all the KV pairs indicated by the pointers in the bottom tier. It continues eviction process if it needs to evict more KV pairs. After the batch eviction, both the evicted tier and pointer structures will be freed. In addition, the server maintains an *expiration bar*, which is a timestamp that indicates the recency of bottom tier. After batch eviction, the expiration bar moves along with the bottom tier and the server sends a message to all the clients about the new expiration bar.

On the other hand, the client keeps an expiration bar for every server. When informed by one server about a new expiration bar, the client updates it accordingly. When the client fetches an expired pointer, it will not use it. However, the client will not invalidate all the expired pointers immediately. Instead, it lazily invalidates them when it needs to insert new pointers to the same entry. This avoids frequent searching of expired pointers. Note that, we do not take the same lazy approach for server-side batch eviction because the server needs the freed blocks to be available in its free list. Also, there can be a pair that is accessed by one client at a very early time but by a different client only recently. In that case, the former client may invalidate a pointer incorrectly because the pair may not be evicted yet, so it will conduct an Active KV operation instead of an Direct KV operation.

IV. Evaluation

Our experiments are conducted on two systems. The first one is an in-house cluster of 21 server nodes called *Innovation*. Each machine is equipped with 10 dual-socket Intel Xeon(R) cores and 64 GB memory. All nodes are connected through an FDR Infiniband interconnect with the ConnectX-3 NIC. The second is the *Titan* supercomputer at Oak Ridge National Laboratory [4]. Titan is a hybrid-architecture Cray XK7 system, which consists of 18,688 nodes and each node is equipped with a 16-core AMD Opteron CPU and 32GB of DDR3 memory. We use the OpenSHMEM implementation in OpenMPI v2.1 both Innovation and Titan. Unless otherwise specified, on each machine we run only one client or server, without collocating them. OpenSHMEM's atomic operation on the Linux cluster is not competitive, so we use the RDMA CAS for exclusive writes on Innovation. We set the number of entries in the set associative hash table to 4096 and the number of entries in the pointer directory to 512.

We evaluate SHMEMCache's performance using both microbenchmarks and the YCSB benchmark suite [10], which generates workloads based on real-world statistics. The microbenchmark consists of 100 thousand operations including either `SET` or `GET` conducted on 1 thousand randomly generated KV pairs of varying sizes. The YCSB workload contains 5 million operations that operate on 5 million records. All records in YCSB workload are generated following Zipfian distribution, with 128 Byte value and a varying `SET`/`GET` ratio. For both benchmarks, we load all the KV pairs in memory before starting the test in order to preclude the impact of data generation. With microbenchmarks, we evaluate the basic latency and throughput of SHMEMCache's `SET` and `GET`. We compare the results to the original Memcached [2] (version 1.4.25) and HiBD, which leverages one-sided RDMA operations for Memcached [1]. Unlike SHMEMCache which uses one-sided `shm_get` for `GET` and one-sided `shm_put` for `SET`, the Memcached server in HiBD uses RDMA write to store the `GET` result to the client, and RDMA read to retrieve the KV pair from the client for `SET` [18]. With YCSB workloads, we evaluate the system throughput, the PDF of operation latency and hit ratio of cached pointers. For each result, we test three times and report the average number.

A. Basic Performance Comparison

We have measured the latency and throughput of SHMEMCache's `SET` and `GET` operations. For latency evaluation, we compare it against the existing Memcached and HiBD. For throughput evaluation, we only compare it against Memcached because the publicly available HiBD package does not provide the functionality for testing throughput performance. The results are shown in Fig. 6.

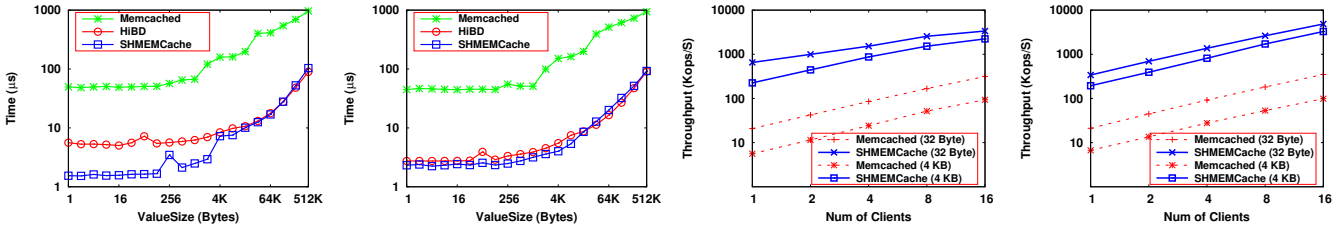
1) Latency: To test the latency, we measure the average time for each operation of varying sizes between one server and one client. As shown in Fig. 6(a) and Fig. 6(b), HiBD and SHMEMCache have an order of magnitude lower latency than the TCP/IP-based Memcached because of their use of RDMA and OpenSHMEM one-sided communication, respectively. Compared to HiBD, SHMEMCache can achieve better latency for small-sized `SET`. This is partly because the RDMA write (used internally for OpenSHMEM's `shm_put`) has better latency than RDMA read for small messages [19], [20]. Our one-sided `shm_put` can leverage the benefit of RDMA write while the overhead of two small CAS operations is not in the critical communication path and therefore overlapped with the main operations. For large-sized `SET` operations, SHMEMCache and HiBD have comparable performance because of the comparable performance from `shm_put` and RDMA write, respectively. However, we can see that SHMEMCache has comparable or slightly lower latency than HiBD for small-sized `GET`. This is mainly due to the fact that SHMEMCache's `GET` completely avoids the server's involvement. Such benefit even offsets the performance disadvantages of small-sized RDMA read to write. Similar to large-sized `SET` operations, large-sized GET operations of SHMEMCache and HiBD have comparable performance. On average, SHMEMCache can provide 128% and 2,534% lower latency for `SET` and 16% and 2,045% lower latency for `GET` compared to HiBD and Memcached, respectively.

2) Throughput: In the throughput test, we use one server but increase the number of clients to up to 16, one per node. We report the results for small 32-byte KV pairs as well as larger 4-KB pairs. As shown in Fig. 6(c) and Fig. 6(d), SHMEMCache outperforms Memcached by at least an order of magnitude for all test cases. We observe that SHMEMCache scales well with the number of clients for both `SET` and `GET`. 32-byte `SET` does not scale as well as 4KB `SET` because the impact of lock contention is more pronounced for smaller SET operations with more clients. As for `GET`, SHMEMCache does not have any significant performance decrease with the increasing number of clients. Overall, SHMEMCache provide 19x and 33x higher throughput for 32-Byte and 4-KB `SET`, and 14x and 30x higher throughput for 32-Byte and 4-KB `GET` than Memcached, respectively.

B. Impact of Different Communication Mechanisms

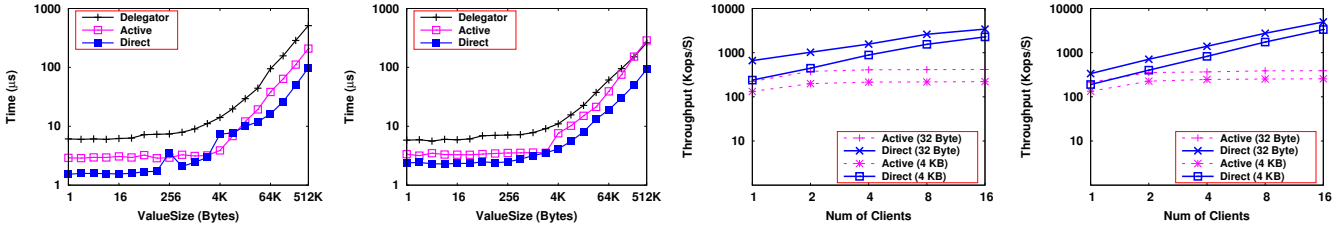
We evaluate the performance of SHMEMCache while enabling either Direct or Active KV operations. In addition, for the latency experiment, we add results of the delegator mechanism into comparison. The delegator mechanism, as mentioned in Section III-A, leverages the messaging conduit for Active `SET`/`GET` to enhance the communication efficiency of the original Memcached. In the remaining sections, we use **Direct**, **Active** and **Delegator** to denote the three communication mechanisms. To test with Direct only, we cache all the pointers at the client side using a larger pointer directory before testing. The results are shown in Fig. 7.

1) Latency: From Fig. 7(a) and Fig. 7(b), we can see that both Direct and Active have significantly better performance than Delegator. Even using the same messaging conduit, Active still outperforms Delegator for most of the cases. This is mainly because Delegator needs to transfer data through shared memory. When comparing between Active and



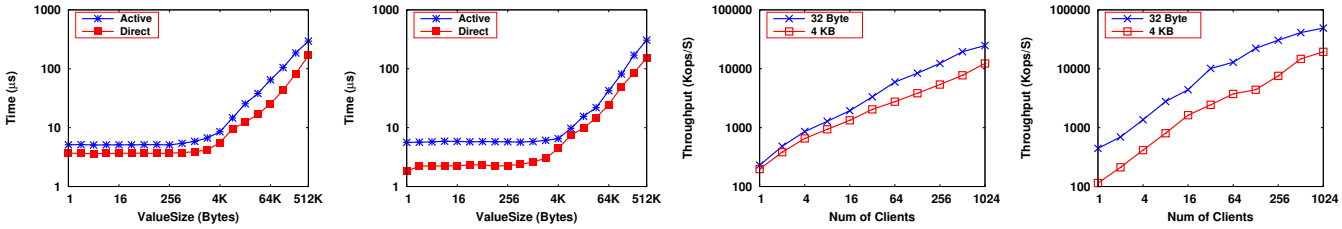
(a) SET latency. (b) GET latency. (c) SET throughput. (d) GET throughput.

Fig. 6: Basic performance comparison.



(a) SET latency. (b) GET latency. (c) SET throughput. (d) GET throughput.

Fig. 7: Impact of communication mechanisms.



(a) SET latency. (b) GET latency. (c) SET throughput. (d) GET throughput.

Fig. 8: Performance on Titan.

Direct, Direct shows a small amount of performance advantage compared to Active, which is mainly due to the avoidance of server-side data polling and processing. For some value sizes (e.g., 256 Byte), Direct SET is slower, which is closely related to the performance variance of shm_put on Innovation. We can also see that the latency of Direct has a trend similar to the overall latency as shown in Fig. 6(a) and Fig. 6(b). This is because a majority of the operations in overall latency test are conducted through Direct because of pointer caching.

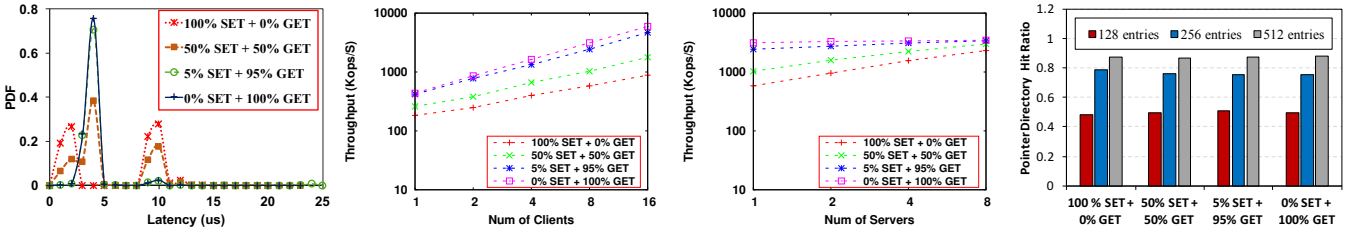
2) Throughput: Same as the basic performance evaluation, we report both small (32 Byte) and large (4 KB) sizes for throughput results. As shown in Fig. 7(c) and Fig. 7(d), Direct has much higher throughput than Active in all cases because it does not need server processing and facilitates more concurrent operations from multiple clients. When using Direct only, SHMEMCache scales well with a larger number of clients. In comparison, when using Active only, a server is saturated at the scale of only 4 clients. This indicates that a server is easily bottlenecked when every client relies on the server to process their operations, especially in a single-thread server design such as SHMEMCache. While we cannot completely remove the Active mechanism, because it helps us avoid many tricky problems and simplify SHMEMCache's design, it is important to minimize its usage in order to achieve higher throughput at large scale.

C. SHMEMCache on Titan

To study the portability and scalability of SHMEMCache on the leadership computing facilities, we deploy it and evaluate its performance on the Titan supercomputer. The results are shown in Fig. 8.

1) Latency: For the latency test, we directly demonstrate the performance of the two communication mechanisms to gain better insight. As we can see from Fig. 8(a) and Fig. 8(b), Direct still outperforms Active for most cases, which is similar to the results on Innovation (Fig. 6). However, we observe much less performance variations on Titan than on Innovation. A comparison between Fig. 8(a) and Fig. 7(a) shows that Direct on Titan has higher latency for smaller SET than on Innovation. This is largely due to the different atomic operations we choose to use on the two platforms as mentioned previously. Moreover, Active on Titan performs slightly worse than on Innovation which is mainly because of the difference in the interconnect speed.

2) Throughput: Since Active has poor scalability for throughput even with only 4 clients, on Titan, we evaluate the throughput of SHMEMCache without differentiating the two mechanisms. We have scaled the number of clients to



(a) PDF of operation latency. (b) Throughput of varying clients. (c) Throughput of varying servers. (d) Hit ratio of pointer directory.

Fig. 9: Performance for YCSB workloads.

up to 1024. As shown in Fig. 8(c) and Fig. 8(d), even with 1024 clients, a single server is still not saturated for both small- and large-sized `SET` and `GET`. Compared to the reported results from two works that also optimize Memcached with one-sided communication operations on HPC systems [7], [17], SHMEMCache shows better scalability in throughput. Moreover, a comparison between Fig. 8(c), Fig 8(d) and Fig. 6(c), Fig. 6(d) shows that SHMEMCache on Titan has a similar trend but slightly lower throughput than on Innovation.

D. Real-world Workloads

We use YCSB to generate four representative types of workloads: `SET`-only, `GET`-only, 95% `GET` + 5% `SET` and 50% `GET` + 50% `SET`. Apart from the latency and throughput, we also evaluate the hit ratio of the pointer directory, which is calculated as follows: number of total operations divided by number of operations that leverage cached pointers. The results are shown in Fig. 9.

1) Latency: We have fixed value size in the YCSB workloads, so instead of varying value sizes, we measure the operation latency and compute the PDF for the latency, as shown in Fig. 9(a). We can see that the distribution of latency for each workloads differs significantly according to their `SET`/`GET` ratio. For workloads that have higher `SET` ratio, there are two convergences around 2 μs and 9 μs, which are mainly the operations conducted by Direct and Active. The `GET`-dominant workloads, on the other hand, have very high convergence around 4 to 5 μs, which is because of the similar performance of Direct and Active for `GET`.

2) Throughput: We also evaluate the system throughput of SHMEMCache under different numbers of servers and clients. Firstly, we fix the number of servers to one and increase the number of clients. As shown in Fig. 9(b), the `GET`-dominant workloads have generally higher throughput and better scalability than the other types that have a smaller ratio of `GET`. This is because the lock contention for popular KV pairs prevents the `SET` throughput from scaling with more clients. Then, we fix the number of clients to eight and increase the number of servers. As shown in Fig. 9(c), introducing more servers significantly boosts the throughput of the all-`SET` workload. This is mainly because more servers can help distribute the lock contention between many clients. But more servers can hardly help boost the throughput of `GET`-dominant workloads. This is mainly because the majority of `GET`s are conducted via Direct which is not constrained by a single server. This also indicates that our cache eviction mechanism can help the client cache popular KV pairs for fast one-sided access.

3) Hit ratio: We measure the hit ratio of the pointer directory but not the hit ratio of the KV store (i.e., how much operations succesfully hit a the targeted KV pair in the store). This is because our objective is not to provide a better cache eviction algorithm but to address the complexities in bringing one-sided operations to KV stores that use LRU for cache eviction. In this test, we vary the number of entries in the pointer directory and test SHMEMCache with different types of YCSB workloads. As shown in Fig. 9(d), the pointer directory has higher hit ratio when its size is larger, which is due to the fact that more space can be used to accommodate more pointers. However, the difference between 256 entries and 512 entries is much less pronounced than the difference between 128 and 256 entries. This is because with the increase in the number of entries, the new entries added will facilitate pointer caching for a collection of less popular KV pairs. Thus, we should carefully tune the size of the pointer directory and not simply expand it as much as possible.

V. Related Works

One-sided operations for in-memory KV stores: Several research efforts have attempted to use one-sided operations such as RDMA write and read to accelerate distributed in-memory KV stores. Appavoo *et al.* [7] and Jose *et al.* [18], [17] both extend Memcached and provide support for RDMA. However, their approaches use RDMA only as a replacement for TCP/IP and still require the server to process all operations. More recently, Pilaf [23], FaRM [11] and HydraDB [29] all optimize `GET` with one-sided RDMA reads that can greatly boost its performance. But their data consistency models are either not friendly to the read-dominant workload, or entail additional complexities that make them less favorable for our purpose (Section II-B). All these prior studies have prevented clients from conducting `SET` directly, which under-utilizes the benefits of RDMA.

Performance enhancement of KV stores: Significant amount of research has been done in the field of performance enhancement of KV stores [5], [12], [14]. FAWN [5] couples low-powered CPUs to allow massively parallel access to data. MemC3 [12] uses Concurrent Cuckoo Hashing to mitigate pointer chasing. MDHIM [14] proposes to use MPI for the communication in a distributed KV store so that it can run on

HPC systems, This resembles one of our purposes in building SHMEMCache, but it mainly targets persistent KV stores. In addition, most of these studies do not touch upon enhancing the communication of KV stores using advanced protocols or techniques such as RDMA.

Cache management: Most of the in-memory KV stores still adopt much simpler policies such as the LRU and its approximations because of their low cost. For example, both Memcached [2] and Redis [3] offer LRU or best-effort LRU. MemC3 [12] uses an LRU approximation based on the CLOCK algorithm. MICA [21] provides both FIFO and LRU. However, because these systems do not use one-sided operations or cache pointers as in SHMEMCache, none of these could address the issues discussed in this paper. Although, C-Hint [28] does address the problem using a lease-based mechanism, this approach adds complexity using by predetermined fixed leases and incurs heavy processing overheads by sending the history report explicitly.

VI. Conclusion

We have proposed to use OpenSHMEM for Memcached to take advantage of OpenSHMEM's efficiency of one-sided communication, visibility of symmetric memory and general portability. We have implemented a prototype called SHMEMCache and synthesized a number of novel design ideas in SHMEMCache in order to address several critical challenges effectively and efficiently. We have evaluated SHMEMCache extensively on both an in-house cluster and Titan supercomputer, and the results show that SHMEMCache provides high-performance KV operations at the scale of 1024 nodes. However, currently, SHMEMCache does not support fault tolerance, e.g., failure of the client that has claimed the write lock may leave the system in an inconsistent state. We plan to address this issue in the future. Also, we intend to investigate various options of hash algorithms and hash table designs for SHMEMCache.

Acknowledgment

We are very thankful for the insightful comments from the anonymous reviewers. This work was supported in part by a contract from Oak Ridge National Laboratory and the National Science Foundation awards 1561041 and 1564647.

References

[1] Hibd. http://hibd.cse.ohio-state.edu/.

[2] Memcached. https://memcached.org/downloads.

[3] Redis. http://redis.io/.

[4] Titan Supercomputer. https://www.olcf.ornl.gov/titan/.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[6] C. Aniszczyk. Caching with twemcache, 2012.

[7] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 385–394. ACM, 2010.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[9] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[11] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[12] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[13] H. Fu, K. Singharoy, M. Gorentla Venkata, Y. Zhu, and W. Yu. Shmemcache: Enabling memcached on the openshmem global address model. In *Workshop on OpenSHMEM and Related Technologies*. Springer, 2016.

[14] H. Greenberg, J. Bent, and G. Grider. Mdhim: a parallel key/value framework for hpc. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[15] J. Jose, K. Hamidouche, X. Lu, S. Potluri, J. Zhang, K. Tomko, and D. K. Panda. High performance openshmem for xeon phi clusters: Extensions, runtime designs and application co-design. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 10–18. IEEE, 2014.

[16] J. Jose, S. Potluri, K. Tomko, and D. K. Panda. Designing scalable graph500 benchmark with hybrid mpi+ openshmem programming models. In *International Supercomputing Conference*, pages 109–124. Springer, 2013.

[17] 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 *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 236–243. IEEE, 2012.

[18] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, et al. Memcached design on high performance rdma capable interconnects. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 743–752. IEEE, 2011.

[19] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[21] 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 (NSDI 14)*, pages 429–444, 2014.

[22] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[23] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[25] S. Pophale, R. Nanjegowda, T. Curtis, B. Chapman, H. Jin, S. Poole, and J. Kuehn. Openshmem performance and potential: A npb experimental study. In *The 6th Conference on Partitioned Global Address Space Programming Models (PGAS12)*. Citeseer, 2012.

[26] S. Potluri, D. Bureddy, H. Wang, H. Subramoni, and D. K. Panda. Extending openshmem for gpu computing. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1001–1012. IEEE, 2013.

[27] D. Shankar, X. Lu, N. Islam, M. Wasi-Ur-Rahman, and D. K. Panda. High-performance hybrid key-value store on modern clusters with rdma interconnects and ssds: Non-blocking extensions, designs, and benefits. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 393–402. IEEE, 2016.

[28] Y. Wang, X. Meng, L. Zhang, and J. Tan. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[29] Y. Wang, L. Zhang, J. Tan, M. Li, Y. Gao, X. Guerin, X. Meng, and S. Meng. Hydradb: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In *SC'15*, page 22. ACM.