High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits*

Dipti Shankar, Xiaoyi Lu, Nusrat Islam, Md. Wasi-ur-Rahman, Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
Email: {shankard, luxi, islamn, rahmanmd, panda}@cse.ohio-state.edu

***Abstract*—High-performance, distributed key-value store-based caching solutions, such as Memcached, have played a crucial role in enhancing the performance of many Online and Offline Big Data applications. The advent of high-performance storage (e.g. NVMe SSD) and interconnects (e.g. InfiniBand) on modern clusters has directed several efforts towards employing 'RAM+SSD' hybrid storage architectures for key-value stores running over RDMA, in order to achieve high data retention, while maintaining low latency and high throughput. In this paper, we first perform a detailed analysis of the behavior of hybrid Memcached designs, and identify two major bottlenecks: the client-side wait for request completion and the server-side SSD I/O overhead. Based on this analysis, we propose new non-blocking API extensions for Memcached Set and Get operations, to support high data retention while trying to achieve near in-memory speeds. We enhance the existing runtime designs on both the client and the server, and propose an adaptive slab manager with different I/O schemes for higher throughput. We demonstrate that Libmemcached-based applications can achieve high performance by exploiting the communication/computation overlap that is made possible by the proposed non-blocking API extensions, with either In-memory or SSD-assisted designs of RDMA-based Memcached. Performance evaluations show that the proposed extensions and designs can achieve up to 16x improvement for Memcached Set/Get latency over current hybrid design for RDMA-Memcached when all data does not fit in memory, and up to 3.6x improvement over pure in-memory design of default Memcached over 'IP-over-IB' when all data can fit in memory.**

***Keywords*-Memcached; SSD; RDMA; Non-Blocking;**

I. Introduction

High-performance, in-memory key-value stores, like Memcached, are being widely used to speed up Online Data processing, including traditional OLTP, web-oriented services, and Cloud-based NoSQL stores. By taking advantage of the high-performance key-value stores, these workloads can alleviate the load on the underlying database, and achieve higher efficiency. Several recent studies in the literature [5, 18, 19, 21] have analyzed read-only and read-heavy workloads running over key-value stores, and proposed several optimized designs [10, 13]. Recently, several Offline Big Data analytical workloads have started to explore the benefits of leveraging high-performance key-value stores. Prominent examples include a Memached-based Burst Buffer layer for the Lustre parallel file systems [20], a Memcached-based Burst-buffer for HDFS [9], and intermediate data caching with Memached in Hadoop MapReduce [22]. These workloads are usually write-heavy, and thus include different I/O and communication characteristics as compared to the traditional Memcached usage scenario.

In general, all these studies propose designs that can leverage a high-performance key-value store-based caching layer to enhance their application performance. Irrespective of whether they are online data processing workloads, offline data analytics, or I/O from scientific computing workloads, we can clearly see that high-performance key-value stores play a crucial role in accelerating these data-intensive applications. In addition to this, the advent of high-performance storage (e.g. SATA SSD, NVMe SSD) and interconnect (e.g. InfiniBand) on modern High-performance compute (HPC) clusters has directed significant efforts towards leveraging a 'RAM+SSD' hybrid storage, to increase the data retention in existing in-memory key-value stores.

A. Motivation

Due to the high demand for high-performance key-value stores, several advanced designs have been proposed to enhance the performance of key-value stores such as Memcached, from the perspective of communication and I/O. Table I gives a brief overview of some of the representative works on different Memcached designs from literature, and shows the main differences between this paper and existing studies. As shown in Table I, since default Memcached (**IPoIB-Mem**) could not take full advantage of high-performance interconnects, such as InfiniBand, through the use of TCP/IP protocol over InfiniBand (i.e., IP-over-IB protocol or IPoIB), RDMA-based Memcached (**RDMA-Mem**) was proposed in [10] to leverage the native RDMA protocol for optimal performance. On the other hand, to minimize the loss of data due to eviction in 'all-in-memory' key-value store designs such as **IPoIB-Mem** and **RDMA-Mem**, and the corresponding performance degradation incurred by the need to access data from a backend layer, such as a database in the case of online data processing workloads [18, 19], several hybrid memory designs with SSD support were proposed, including SSD-Assisted Hybrid Memcached [17] (**H-RDMA-Def**) and FatCache [7].

Based on these publicly available distributions, we ran experiments with a variation of the OSU HiBD Benchmark (OHB) [16], using a Zipf request distribution, to investigate

*This research is supported in part by National Science Foundation grants #CNS-1419123, #ACI-1450440 and #IIS-1447804. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.112

Design Features		IPoIB-Mem[3]	RDMA-Mem[10]	FatCache[7]	H-RDMA-Def[17]	This Paper
RDMA-based Communication		N	Y	N	Y	Y
Hybrid Memory with SSD	Basic	N	N	Y	Y	Y
	Adaptive I/O enhancements	N	N	N	N	Y
	NVMe-SSD	N	N	N	N	Y
Non-Blocking API Extensions		N	N	N	N	Y

Table I
DESIGN COMPARISON WITH EXISTING WORK

the performance characteristics of different Memcached designs on a modern HPC cluster (SDSC Comet [4]) with InfiniBand FDR interconnects and SATA SSDs. As shown in Figure 1(a), when data fits in memory, RDMA-based designs (both RDMA-Mem and H-RDMA-Def) can outperform default Memcached running with IPoIB protocol **IPoIB-Mem**, in terms of average Set/Get latency. However, when the data does not fit in memory, the overall latency is subjected to additional caching layer miss penalty, due to the need to access a backend database. For **IPoIB-Mem** and **RDMA-Mem**, we assume a penalty of less than 2 ms. As seen in Figure 1(b), since **H-RDMA-Def** design can hold more data in its hybrid memory space, the overall latency is considerably reduced. This clearly illustrates the benefits of hybrid memory with SSDs for Memcached.

However, if we compare the latency numbers of the **H-RDMA-Def** across Figure 1(a) and Figure 1(b), we can see a **15-17x** performance degradation due to SSD access. This scenario motivates us to answer a challenging question: **Can we significantly improve the performance of the existing hybrid Memcached design with RDMA and SSDs by orders of magnitude, so that the overall performance can be as close as possible to the performance when all data fits in memory?**

This further leads to the following broad challenges:

- Where are the fundamental bottlenecks in the existing hybrid Memcached design? How can we find them?
- Can we propose new designs and approaches to improve the performance of RDMA-enabled hybrid Memcached with SSD support, by orders of magnitude?
- Can these proposed designs achieve the optimal performance of the existing designs when data fits in memory, even when available memory does not facilitate caching all data for high-speed access?
- If so, how much performance benefits can be achieved for Memcached-assisted workloads, by exploiting RDMA and various high-performance SSDs, such as SATA or NVMe SSDs, on modern HPC clusters?

B. Contribution

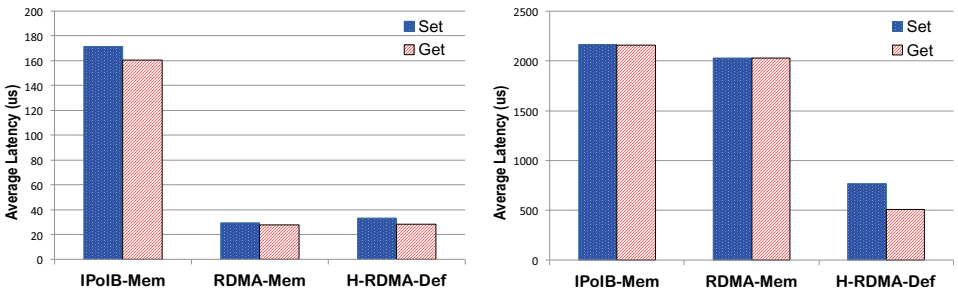
In order to achieve high data retention, while maintaining low latency and high server throughput, we need to study the communication and I/O performance characteristics of Memcached. In this paper, we first perform a detailed analysis of the behavior of different Memcached designs. We identify different critical stages in the Memcached Set/Get operations, and find that (1) waiting time on the client side and (2) SSD access times on the server side are the major bottlenecks currently. With this as the basis, we introduce non-blocking extensions to the existing Libmemcached APIs, that can be utilized to significantly improve the performance of **both in-memory and hybrid RDMA-based Memcached designs**. We also propose adaptive I/O enhancements inside the hybrid Memcached slab management layer, that can improve the performance of even the default blocking Set/Get APIs by up to **3x**. Through our extensive performance studies, we find that our proposed designs and extensions can improve the Set/Get latency by up to **16x**, as compared to the current hybrid RDMA-based Memcached designs; thus making the average operation latencies achievable by the hybrid design to be very close to that of pure in-memory RDMA design. We demonstrate that we can **leverage the computation/communication overlap on the client-side** to hide the bottlenecks that contribute to the long data access times, and improve the throughput of the hybrid RDMA-based Memcached design by up to 2.5x.

To the best of our knowledge, this is the first work that proposes non-blocking APIs for Memcached with operation completion guarantees, that can co-exist with the current blocking APIs, and investigates their performance benefits with RDMA and SATA-/NVMe-SSDs. The following describes the organization of the paper. Some background knowledge and related works are discussed in Section II. Section III analyzes the performance bottlenecks in existing designs. Section IV discusses the proposed non-blocking APIs for Memcached. Section V presents our runtime design. In Section VI, we present our evaluations. We conclude the paper in Section VII with future works.

II. BACKGROUND AND RELATED WORK

A. Memcached Distributions and Large-Scale Deployments

Memcached is a distributed memory caching system proposed by Fitzpatrick [8], which allows a group of intermediate servers to pool their memory resources together, to cache frequently accessed data objects (e.g. database queries, results of API calls etc.), and minimize the number of times the actual data is read from a back-end data store or a database. Memcached follows key-value semantics where each client can store and retrieve items from the servers using "keys". Workloads that depend on Memcached are extremely latency sensitive. Even though the underlying architecture of Memcached is scalable, the performance of Memcached is directly related to that of the underlying networking technology. The benefits are also tightly coupled



(a) Data Fits In-Memory (b) Data Does Not Fit In-Memory (With Penalty)

Figure 1. Overall Latency For Memcached Set/Get Operations

with the aggregated amount of RAM available, which determines how often the database shall be queried. Memcached is widely used by Twitter, Facebook, Youtube etc., to speed up large-scale query processing.

B. Other RDMA-enhanced Key-value Stores

In-memory, key-value stores have been successfully leveraging high-performance interconnects, by employing protocols like RDMA over InifiniBand [10]. Along similar lines, HERD [11] is an RDMA-based key-value store that is designed for reducing network round trips, while using efficient RDMA primitives, and Pilaf [15] is a key-value store design that uses one-sided RDMA to achieve high performance with low CPU overhead. MICA [13] is another scalable RDMA-based in-memory key-value store that is optimized for multi-core architectures. MICA takes a holistic approach to handle both read- and write-intensive workloads at low overhead.

C. Flash-storage Assisted Key-Value Stores

Flash-storage Assisted Key-Value Stores are most suited for large workloads that need to maintain high hit rate for high performance. Fatcache [7] from Twitter extends a volatile, in-memory cache by incorporating SSD-backed storage, while performing minimal disk reads on cache hits and random disk writes. SILT[12] is another memory-efficient, high-performance key-value store system that combines new algorithmic and systems techniques to balance the use of memory, storage, and computation, to provide comparable throughput. CaSSanDra [14] is an SSD boosted key-value store that extends commercialized Cassandra key-value store engine to include SSD in its storage memory. Similarly, McDipper [2] is a high performance, flash-based cache server that is Memcached protocol compatible, which is in active use at Facebook. In this paper, we explore new opportunities for improving the performance of RDMA-based Memcached designs.

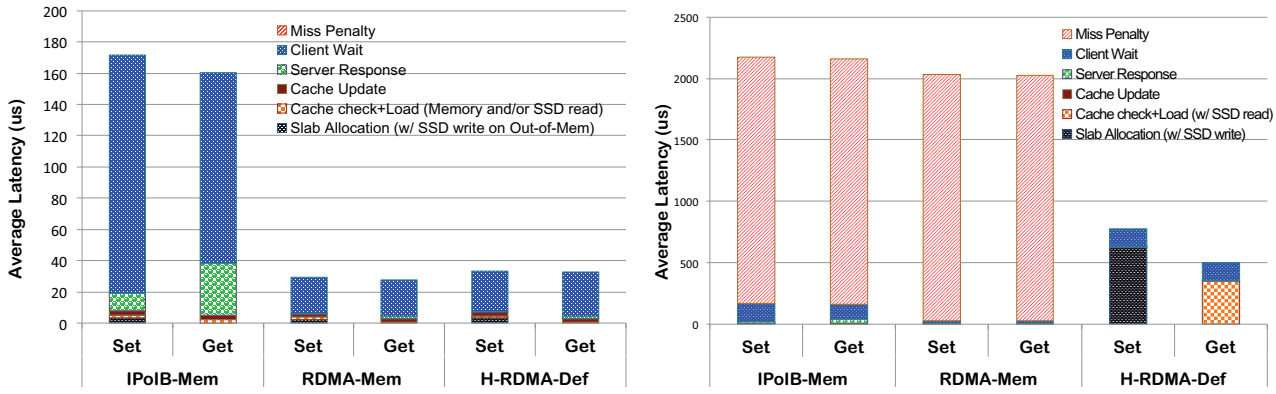
III. Bottleneck Analysis in RDMA-enhanced Hybrid Memcached

From Figure 1, we observe two aspects of the current SSD-assisted RDMA-based Memcached design [17]:
(1) The performance is similar to that of the in-memory design, when all data fits in memory. This indicates that the hybrid design incurs negligible overhead when sufficient memory is available.
(2) When all data does not fit into memory, the performance improvement is significant compared to the in-memory designs, that incur the overhead of accessing a much slower backend layer. For instance, in typical web-scale workloads, every miss in the Memcached server leads to a query being issued to the underlying database.

However, from Figures 1(a) and 1(b), we notice that the performance of the current SSD-assisted RDMA-based design can significantly degrade if the miss penalty incurred is negligible. In this section, we first perform an in-depth analysis of the performance bottlenecks in the default Memcached design running 'IP-over-IB' (**IPoIB-Mem**), RDMA-based In-memory Memcached (**RDMA-Mem**) and RDMA-based Hybrid Memcached (**H-RDMA-Def**) designs, and explore the possibilities of improving their performance.

A. Characterization Methodology

We study the time-wise breakdown of the Memcached Set/Get operations in various Memcached designs. We profile the time spent by these operations at six different stages:
(1) Slab Allocation: Memcached employs a slab allocation mechanism to optimize memory usage, and prevent memory fragmentation when information expires from the cache. Memory is reserved in blocks of 1MB, and divided into smaller chunks for items belonging to designated slab class. This memory management stage plays crucial role in retaining and evicting data from the in-memory cache. With respect to the SSD-assisted hybrid design, the slab allocation can involve the flushing of key-value pairs to the SSD using the direct I/O scheme, when available memory is insufficient.
(2) Cache Check and Load: In this phase, the Memcached server checks for the latest version of the requested key-value pair. This phase helps formulate a server response. In the SSD-assisted hybrid design, this phase can involve reading the key-value pair from the SSD.
(3) Cache Update: This phase mainly maintains the freshness of the data in the Memcached servers. It promotes the most recently added or accessed data to the head of the LRU list it maintains.
(4) Server Response: In this phase, the Memcached server prepares the actual response, and communicates the opera-



(a) Data Fits In-Memory

(b) Data Does Not Fit In-Memory

Figure 2. Time-wise breakdown of Memcached Set/Get Latency

tion completion status and/or the data to the client.

(5) Client wait: In this phase, the client issues a request to the Memcached server, and waits on its completion.

(6) Miss Penalty (Backend Data Access): When a key-value pair is not found in the Memcached server cluster, the client issues a data request to the backend database, and caches the results in the Memcached server cluster to speed-up future look-ups.

For our analysis, we use Cluster A (SDSC Comet), that is equipped with SATA SSDs (described in Section VI). We undertake latency tests with a single Memcached server and client, and perform in-depth analysis of the time-wise breakdown of the different stages described above.

B. Time-wise breakdown of Data Access Latency

We study the performance of different Memcached designs running over InfiniBand (IB), using a Zipf-like request distribution that issues repeated requests to a subset of the data. First, we study the time-wise breakdown when all data fits in memory. The Memcached server is preloaded with 1 GB of data (key-value pairs of size 32 KB). From Figure 2(a), we observe that both in-memory designs i.e, **IPoIB-Mem** and **RDMA-Mem**, spend a considerable amount of time communicating data over the network, as compared to the other phases. Specifically, with the RDMA-based design, we observe that the client spends a significant amount of its time waiting for the completion of the issued Memcached Set or Get request. Thus, when all data fits in memory, the network tends to be the major bottleneck.

Next, we study the time-wise breakdown when all data does not fit into memory. The Memcached server is preloaded with 1.5 GB of data (key-value pairs of size 32 KB). For every miss in the Memcached caching layer, we assume that the overhead or penalty is less than 2 ms. From Figure 2(b), we can see that the backend database access is a major contributor to the latency among the in-memory designs i.e, **IPoIB-Mem** and **RDMA-Mem**. The hybrid design **H-RDMA-Def** significantly outperforms the in-memory design, especially when the overhead of accessing the backend storage layer is high. However, if we contrast the performance of **H-RDMA-Def** in Figures 2(a) and 2(b), we observe that the SSD I/O accesses impose a very significant overhead.

C. Summary

To summarize, from Figure 2, we observe that:

- In spite of the very low data access latencies, the client spends a considerable amount of time blocking for the completion of a Memcached Set or Get operation.
- While the SSD-assisted hybrid design for RDMA-based Memcached can guarantee a very high success rate due to its property of high data retention, the cost of SSD I/O incurred in the course of the Set or Get operation cannot be ignored.

Based on these observations, we propose the following extensions to the current SSD-assisted hybrid design for RDMA-based Memcached, to improve Set/Get latencies while retaining high key-value store capacities:

- There is room for improvement with respect to the semantics of the default Memcached APIs. We introduce new non-blocking API extensions to the current Memcached Set and Get operations.
- Since data persistence is not a characteristic of the default Memcached design, we explore the possibilities of leveraging different I/O schemes, beyond direct I/O, to alleviate the SSD access bottleneck in the RDMA-based hybrid Memcached design.

IV. Extending Memcached to Support Non-Blocking Operations

The current hybrid design with RDMA-based Memcached provides support for blocking Memcached Set/Get APIs only, i.e., `memcached_set` and `memcached_get`, which mandates that the client waits until the operations complete at the Memcached server. As discussed in Section III, this could lead to increased wait on the client-side, due to the synchronous SSD I/O overhead incurred on a very busy hybrid Memcached server.

A. Proposed Extensions of Non-Blocking Memcached APIs

Proposed Non-blocking Extensions: We introduce mechanisms to fetch and store data into Memcached in a non-blocking manner. The proposed extensions allow the user to separate the request **issue** and **completion** phases. This enables the application to proceed with other tasks, such as computations or communication with other servers, while waiting for the completion of the issued requests. We benefit from the inherent one-sided characteristics of the underlying RDMA communication engine, and ensure the completion of every Memcached Set/Get request. In this way, we can avoid incurring significant overheads due to client-side blocking waits and server-side SSD I/O.

```
// Non−blocking set. You can NOT reuse the key/value buffers↩
      until either a successful wait/test or you know that ↩
     the key/value has been received in the server side.
memcached_return memcached_iset(memcached_st *ptr, const ↩
     char *key, size_t key_length, const char *value, ↩
     size_t value_length, time_t expiration, uint32_t ↩
     flags, memcached_req *req);

// Non−blocking get. You can NOT reuse the key buffer until ↩
     either a successful wait/test or you know that the key↩
      has been received in the server side.
char *memcached_iget(memcached_st *ptr, const char *key, ↩
     size_t key_length, size_t *value_length, uint32_t *↩
     flags, memcached_req *req, memcached_return *error);

// Non−blocking set. You CAN reuse the key/value buffers ↩
     once this API returns.
memcached_return memcached_bset(memcached_st *ptr, const ↩
     char *key, size_t key_length, const char *value, ↩
     size_t value_length, time_t expiration, uint32_t ↩
     flags, memcached_req *req);

// Non−blocking get. You CAN reuse the key buffer once this ↩
     API returns.
char *memcached_bget(memcached_st *ptr, const char *key, ↩
     size_t key_length, size_t *value_length, uint32_t *↩
     flags, memcached_req *req, memcached_return *error);

// Testing non−blocking API completion
void memcached_test(memcached_st *ptr, memcached_req * req);

// Waiting on non−blocking API completion
void memcached_wait(memcached_st *ptr, memcached_req * req);
```

Listing 1. Non-Blocking API Extensions to libMemcached

We introduce two types of non-blocking Memcached Set/Get APIs, and supporting structures, as shown in Listing 1:
(1) Memcached Request Structure: To facilitate the proposed APIs, we introduce a new structure called `memcached_req` that contains: (1) completion flag that the user can test or wait on for operation completion, (2) pointer to the buffer where the server's response will be available, and (3) pointers to user's key-value pair buffers.
(2) Set/Get API with no immediate buffer re-use guarantee: We introduce `memcached_iset` and `memcached_iget` purely non-blocking APIs to issue Set and Get requests to the Memcached server using RDMA. These APIs do not guarantee that the key-value buffer can be immediately re-used. However, these APIs are faster, as they can return to the user application as soon as the request header has been sent out from the RDMA communication engine.
(3) Set/Get API that guarantee re-use of user buffers: We introduce `memcached_bset` and `memcached_bget` APIs to send non-blocking Set and Get requests to the Memcached server using RDMA. These APIs guarantee that the client's key and value buffers, i.e., `(char *key, char *value)` can be re-used when the non-blocking call returns to the client. Since memory registration is a costly affair with RDMA-enabled interconnects, provisioning buffer re-use is extremely helpful.
(4) Wait/Test API: We introduce `memcached_wait` and `memcached_test` APIs to check the progress of the operation in either a blocking or non-blocking fashion, respectively. Each of these accept a `memcached_req` structure, and return the current status of the operation.

Contrasting Buffers IO mode in default Libmemcached: With respect to Memcached Set operations, the default TCP/IP-based Libmemcached enables the client to use asynchronous IO, and facilitates buffering IO requests. Any action that gets data causes the buffered requests to be sent out to the Memcached server. While these behavioral modes can help accelerate the Set operations, especially for small data sizes, the following Get requests issued incur a considerable overhead, as they need to push out any data remaining in the queue before proceeding. Our proposed non-blocking API extensions are different and more beneficial in the following ways:

1) We propose non-blocking APIs which can co-exist with current blocking APIs. With the current Libmemcached library, setting of the behavior to use buffering and non-blocking mode is imposed on all operations.
2) Instead of just buffering the I/O requests, our proposed API extensions leverage the one-sided communication operations to send out the request quickly, and facilitate the user to asynchronously monitor the key-value pair storage or retrieval operations.
3) We introduce `wait` and `test` APIs to guarantee that the server's response reaches the client, and also enhance the server to support asynchronous request completion. However, the default Libmemcached library and the default Memcached server design does not currently provide an explicit way to guarantee the operation completion in their non-blocking mode.

In this way, the proposed non-blocking API extensions can provide much better flexibility to the end-user applications.

B. Application Example with Proposed Non-Blocking APIs

In this section, we present an example of how the proposed API extensions can be leveraged by the application. As shown in Listing 2, this example emulates the behavior of bursty I/O, similar to the workloads in [9, 20]. Data is read and written in blocks, and each block is divided into chunks that can fit into key-value pairs. These appli-

cations usually guarantee completion on a block-by-block basis, and the chunks of a block can be scattered across multiple Memcached servers. With the non-blocking APIs, the client can quickly issue several memcached_iset or memcached_bset operations for the chunks of the given block, and test their completion in a non-blocking fashion using memcached_test at the end of each data block send. The application then finally waits to ensure that all data blocks have been successfully written into the Memcached servers, using memcached_wait. Similarly, for reading a data block, multiple memcached_iget or memcached_bget operations can be issued for the chunks of the requested block. Similarly, a data block can be read by the client without having to block on each and every chunk, by leveraging memcached_iget or memcached_bget operations. Unlike the default blocking APIs, the client can achieve good overlapping by asynchronously communicating with multiple servers, and improve performance by hiding the SSD I/O overheads in the hybrid Memcached design from the user application.

```
// Saving multiple data blocks with chunks to memcached ↩
     servers
for(i = 0; i< nBlocks; i++) {
  for(j = 0; j < nChunks; j++) {
     sprintf(memc_key, ``BR%.4dB%.4dC", i, j);
     memcached_iset(memc, memc_key, key_sz, chunk_buf, ↩
          chunk_sz, expiration, &flags, &allreqs[i][j]);
  }
  // Done sending all chunks for one block
  for(j = 0; j < nChunks; j++) {
     memcached_test(memc, &allreqs[i][j]);
  }
  // Do some other computations possibly
}
// Wait for all completions. Data transmissions get
// overlapped among different blocks and chunks
for(i = 0; i< nBlocks; i++) {
  for(j = 0; j < nChunks; j++) {
     memcached_wait(memc, &allreqs[i][j]);
  }
}
```

Listing 2. An Example of Bursty I/O Applications Using the Proposed Non-Blocking Memcached APIs

V. Proposed Design For Accelerating Hybrid Memcached For HPC Clusters

In this section, we present the enhancements to current runtime design for supporting the API extensions proposed in Section IV. We also present analysis of adapting different I/O schemes to get the best benefit for all data sizes, especially with the proposed non-blocking API extensions.

A. Enhanced Runtime Design For Non-Blocking Libmemcached API

As discussed in the previous section, we propose two types of non-blocking APIs for Memcached Set/Get operations. While the semantics are similar, they differ in how far down the stack the operation ventures before returning to the client. They facilitate the asynchronous completion of the operations, thus avoiding the overhead that the client would incur if it had to synchronously wait for the receipt of operation completion, involving the costly hybrid memory slab allocation and eviction described in Section III. Figure 3 shows the enhancements made to the Memcached runtime design for accommodating the proposed non-blocking APIs.

1) Supporting Non-Blocking API Semantics: To support the non-blocking memcached_iset and memcached_iget operations, the Libmemcached runtime guarantees that the request issue and completion phases can take place independently. The client library chooses the server based on the hashed key, and passes server information and key-value pair buffers to the underlying communication library. The runtime prepares the header packet containing the key, sends out the request header, and returns to the client immediately. In the background, the underlying communication engine runtime completes the request by sending/receiving the corresponding value to/from the Memcached server. Both memcached_iset and memcached_iget return a completion flag that the client can use to guarantee the actual Set/Get request completion. While we do not guarantee that the key-value buffers can be re-used when the API call returns, we accelerate the operation completion from the client's perspective. This path is depicted in violet in Figure 3.

2) Guaranteeing Re-usable Key-Value Buffers At Client: To support the memcached_bset and memcached_bget operations, the Libmemcached runtime needs to guarantee that the key-value buffers can be re-used by the client. For memcached_bset, a request is issued to the underlying RDMA communication engine, and the header containing key is passed to it, along with the value buffer. It then waits for the underlying RDMA communication engine to communicate that it has sent out the data, and the buffer is free for re-use. For memcached_bget, a request is issued to the underlying RDMA communication engine, and the header containing key is passed to it, along with the pointer to where the fetched value needs to be copied. It then waits for the underlying RDMA communication engine to send out the header. The runtime returns the completion flag on which the client can wait to guarantee the actual completion of the operation as necessary. In this way, we can ensure that the user-provided buffers are reusable when the API completes its execution. The non-blocking APIs return a completion flag which the clients can wait on or test, to check if the actual communication involved in the Set or Get is complete. This path is depicted in red in Figure 3.

3) Communication Completion: The functions memcached_wait and memcached_test are used to complete a non-blocking communication. The memcached_wait operation is a blocking call that waits at the underlying RDMA communication engine until it receives the Memcached server's response. In contrast, the memcached_test is a non-blocking call that checks

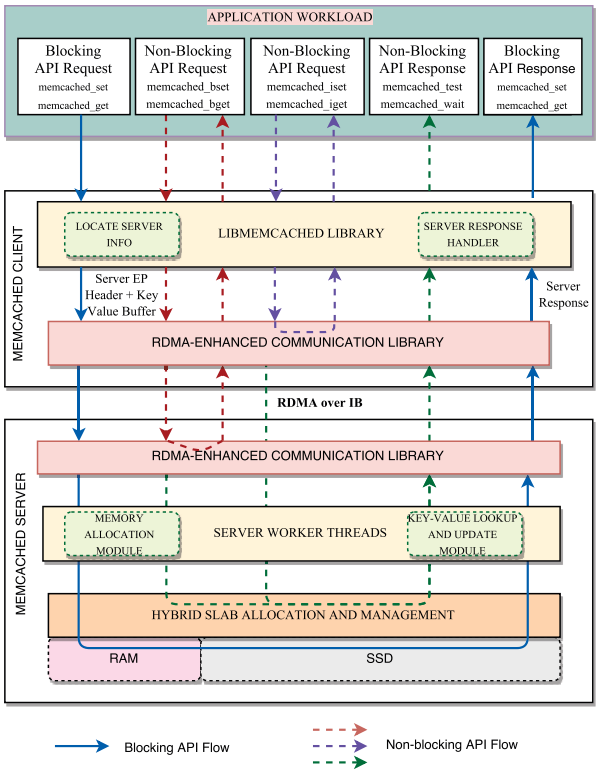


Figure 3. New Memcached API Design with Non-Blocking Operations

with the underlying RDMA communication engine if the server's response is available. The completion of an `iset` or `bset` operation indicates that the key-value pair has been successfully or unsuccessfully stored in the Memcached server pool. The completion of an `iget` or `bget` operation indicates that the key-value pair requested has been fetched from the server and copied into the user buffer, or the server response to a non-existing key was received. With respect to `iget`/`iset` operations, the sender buffer is now free for re-use. This path is depicted in dark green in Figure 3.

B. Enhancements to Hybrid Memcached Server

1) Facilitating Non-Blocking API On The Server: In order to support the non-blocking API extensions, we enhance the server-side request handling phase by separating the memory access/update phase and communication phase. The server buffers the client's request and data, and notifies the client that its buffer can be re-used. This is especially helpful when accelerating the `bset` operation. The server then proceeds with slab/memory management and cache update phases, for each of the buffered requests, and communicates the success or failure of the key-value pair store or fetch operation to the appropriate client. In this way the expensive hybrid memory eviction and updating can happen asynchronously, while the client can proceed with other operations with the same or different servers in the mean time. This is indicated by the dotted green path on the Memcached server side in Figure 3.

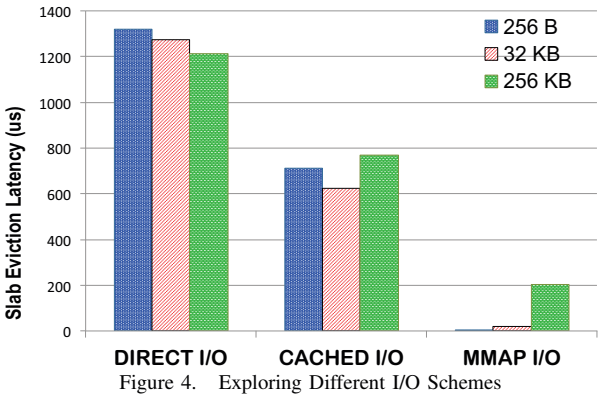


Figure 4. Exploring Different I/O Schemes

2) SSD I/O Data Manangement Schemes: The existing RDMA-based hybrid Memcached design employs the same I/O scheme for all data sizes, and this can significantly impact the server's throughput. On each failed attempt to allocate memory in the RAM slabs, an entire slab (usually 1 MB) of key-value pairs is buffered and synchronously flushed to the SSD. This imposes a large I/O penalty even for small data sizes. In addition to extending support for the non-blocking API semantics, we explore how best to perform synchronous eviction of cached data to the SSD, for different data sizes. We explored different methods of performing synchronous I/O including direct I/O, cached I/O, and mmap, in Figure 4. From our experiments, we observe that maximum benefits can be achieved when evicted data employs (1) mmap-ed slabs for small data sizes, and, (2) normal cached I/O over direct I/O for large data sizes. Based on this, we redesign the slab allocator in Figure 3 to adaptively choose slabs for eviction for best performance, as shown in Figure 5. Based on the slab class chosen for a given key-value pair, the slab allocator switches between mmaped-I/O and cached I/O, when data is being evicted to the SSD due to insufficient memory slabs.

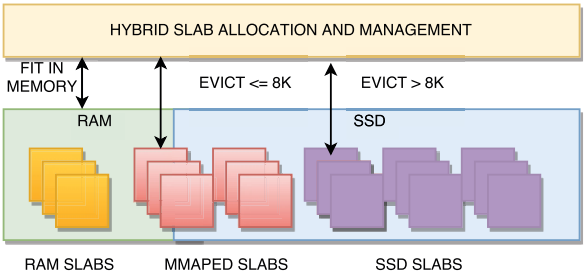


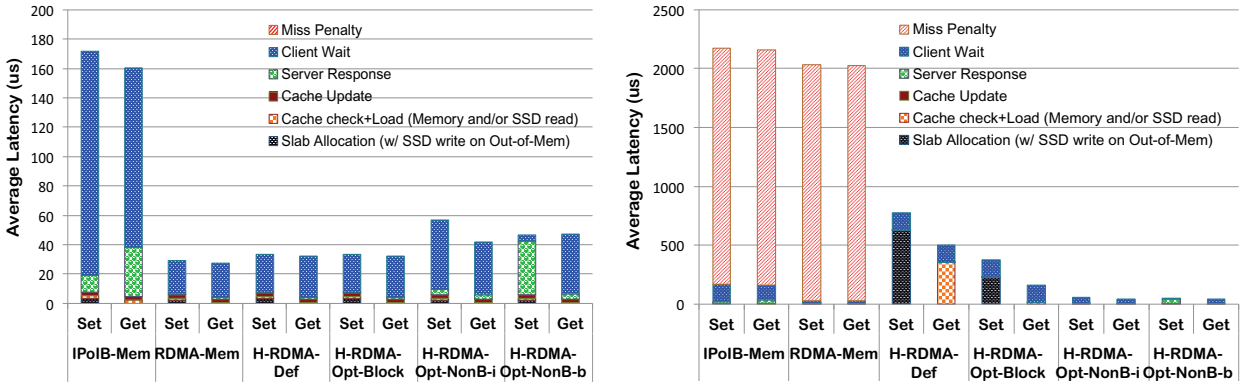
Figure 5. Adaptive Slab Allocation and I/O

VI. Performance Analysis on HPC Clusters

In this section, we present the evaluation methodology and the results of our in-depth analysis of the benefits of the proposed non-blocking APIs and optimizations.

A. Evaluation Methodology and Micro-benchmarks

We employ the micro-benchmarks for Memcached presented in [19], designed to mimic web-scale workloads that



(a) Data Fits In-Memory (b) Data Does Not Fit In-Memory

Figure 6. Time-wise breakdown of Hybrid RDMA-Memcached Set/Get Latency with Blocking and Non-Blocking APIs

can leverage a high-performance Memcached-based caching layer. These micro-benchmarks support different data access distributions, that are commonly found in cloud-based workloads like YCSB [6], including Uniform and Zipf-based skewed access patterns. We extend these micro-benchmarks to include a block-based I/O pattern, that read and write blocks of data by dividing them into smaller chunks, to mimic bursty I/O workloads. The chunk or key-value pair size, the overall workload size, data access pattern, workload operation mix (read:write operations per client) can be easily configured to study different dimensions. We have modified these micro-benchmarks to support non-blocking operations. We issue a large iteration of non-blocking Set/Get requests, and wait at the end for all the operations to complete. Based on this micro-benchmark suite, we divide the evaluations into three categories:

(1) Extending evaluations presented in Section III to compare current In-memory and SSD-assisted RDMA-based designs, with the extensions proposed in Section V.

(2) Studying the impact of different data sizes and the overlap percentage available in the user application.

(3) Performance evaluations with NVMe/SATA SSDs for exploring the advantages of leveraging the proposed non-blocking extensions for bursty I/O-like workloads.

B. Experimental Setup

We use the following two cluster for our evaluations,

(1) **SDSC Comet (Cluster A)**: Each compute node in this cluster has two twelve-core Intel Xeon E5-2680 v3 (Haswell) processors, 128GB DDR4 DRAM, and 320GB of local SATA-SSD with CentOS operating system. The network topology in this cluster is 56Gbps FDR InfiniBand with rack-level full bisection bandwidth and 4:1 over-subscription cross-rack bandwidth.

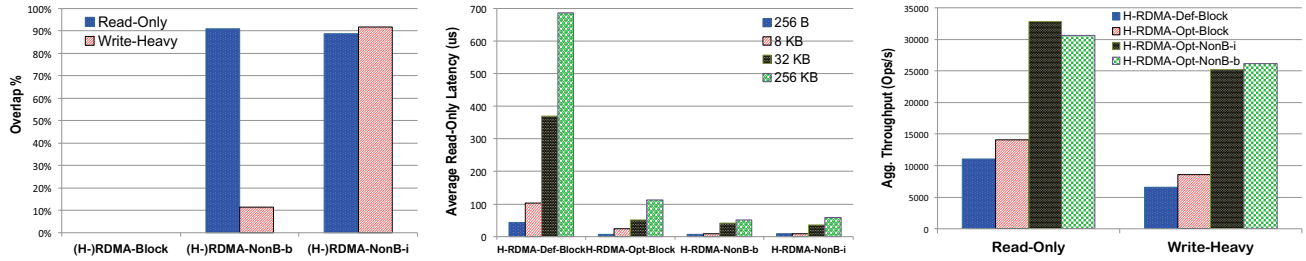
(2) **OSU NowLab (Cluster B)**: Each of them has dual ten-core 2.6 GHz Intel Xeon E5-2660 v3 (Haswell) processors, 64 GB main memory and is equipped with Mellanox 56Gbps FDR InfiniBand HCAs with PCI Express Gen3 interfaces. The nodes run Red Hat Enterprise Linux Server release 6.5. We choose three nodes from this cluster that are equipped with Intel P3700 NVMe-SSD.

We use up to 32 nodes on Cluster A, to emulate varying number of clients, For default Memcached, we use Memcached server version 1.4.24 [3] and libmemcached version 1.0.18 [1]. For RDMA-based designs, we use In-memory and SSD-assisted RDMA-based Memcached version 0.9.3 [16]. Using the micro-benchmark suite described in Section VI-A, we present evaluations with the proposed server-side optimizations using blocking APIs **(H-RDMA-Opt-Block)**, the proposed non-blocking extensions that guarantee buffer re-use **(H-RDMA-Opt-NonB-b)** using `bset` and `bget` operations, and the proposed purely non-blocking extensions **(H-RDMA-Opt-NonB-i)** using `iset` and `iget` operations, in addition to default Memcached running over IPoIB **(IPoIB-Mem)**, In-memory RDMA-based Memcached **(RDMA-Mem)**, and existing SSD-assisted RDMA-based Memcached **(H-RDMA-Def)**.

C. Overall Performance Evaluation

In this section, we extend the evaluations performed in Figure 2, to include the proposed non-blocking extensions and related optimizations. For the first test, when all data fits into memory, a Memcached server is preloaded with 1 GB of data (with key-value pairs of size 32 KB). We measure the latency using a single client that issues Set/Get operations to a Memcached server, using a skewed Zipf distribution. From Figure 6(a), we observe that the proposed non-blocking extensions for the hybrid design, i.e., **H-RDMA-Opt-NonB-i** and **H-RDMA-Opt-NonB-b**, can achieve performance similar to that of the in-memory design **RDMA-Mem**.

For the second experiment, the Memcached server configured with 1 GB RAM is preloaded with key-value pairs of size 32 KB, to a total of 1.5 GB. The In-memory designs evict items based on an LRU policy, while the hybrid designs retain all data in hybrid memory. From Figure 6(b), we observe that **H-RDMA-Opt-Block** can deliver up to 2x improvement over **H-RDMA-Def**, by taking advantage of the adaptive I/O schemes. Most importantly, we can



(a) Overlap% with Different Workload Patterns (b) Performance with Varying Key-Value Sizes (c) Scalability with Non-Blocking API

Figure 7. Overlap% & Key-Value Pair Sizes

see that **H-RDMA-Opt-NonB-i** and **H-RDMA-Opt-NonB-b** can give up to **10-16x** improvement over **H-RDMA-Def**, and up to **3.3-8x** improvement over **H-RDMA-Opt-Block**. Across Figures 6(a) and 6(b), we observe that the non-blocking extensions **H-RDMA-Opt-NonB-b/i** can improve latency by up to 3.6x over **IPoIB-Mem**. Thus the proposed enhancements can benefit Memcached-assisted applications irrespective of whether or not all data can fit in memory.

D. Overlap with Non-Blocking API

In this section, we illustrate that the asynchronous nature of the non-blocking extensions can provide us with the opportunity to exploit overlapping in the user application. We perform this experiment using a single hybrid Memcached server with 1 GB memory and a Memcached client, with 32 KB key-value pair size, and an overall data size of 1.5 GB. We measure the overall percentage of job's runtime that is available for overlap. We run both read-only (100% read) and write-heavy (50:50) workload patterns, using a Zipf request distribution. From Figure 7(a), we observe that **RDMA-NonB-i** (`iset/iget`) can provide up to 92% for both types of workloads, while **RDMA-NonB-b** (`bset/bget`) can provide up to 89% overlap for read-only workloads. We also observe that the overlap% for write-heavy workloads is less than 12% for **RDMA-NonB-b**, as the client needs to block in order to ensure buffer re-usability. All these are in contrast with the current RDMA-based Memcached using blocking APIs (**RDMA-Block**), that offers no overlap. We also ran similar tests with key-value pairs of varying sizes. From Figure 7(b), we observe that our proposed optimizations (**H-RDMA-Opt-NonB-i/b**) can improve performance by about 65-89% over **H-RDMA-Opt-Block** and **H-RDMA-Def-Block**.

E. Scalability with Non-Blocking API

In this section, we illustrate the significant performance benefits achievable in terms of the aggregated server throughput in Operations/second. We perform this experiment with multiple clients, i.e, 100 clients running on 32 nodes, issuing concurrent Set/Get requests using a Zipf-like skewed distribution. We use 4 Memcached servers with an aggregated memory of 1 GB, and SSD usage is limited to 4 GB. We pre-load the Memcached servers with 2 GB of data, constituted by 8 KB key-value pairs.

From Figure 7(c), we observe that **H-RDMA-Opt-NonB-b** and **H-RDMA-Opt-NonB-i** can achieve up to 2-2.5x improvement in throughput over the blocking designs, i.e., **H-RDMA-Opt-Block** and **H-RDMA-Def-Block**. We also observe that the adaptive I/O schemes used in **H-RDMA-Opt-Block** can improve the throughput by about 1.3x over the current direct I/O-based design of **H-RDMA-Def-Block**.

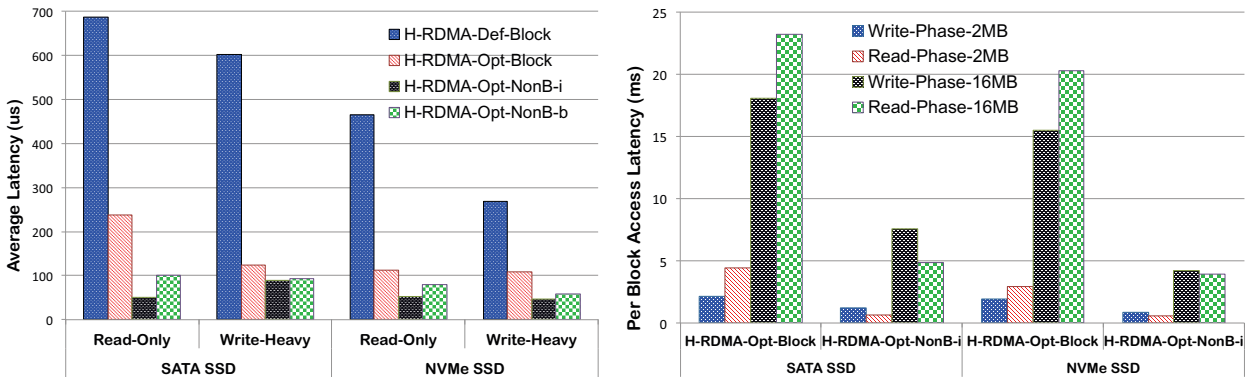
F. Evaluation with NVMe SSDs

In this section, we present performance evaluations with the SSD-Assisted hybrid design for RDMA-based Memcached running over NVMe SSDs (on Cluster B), and contrast it with the performance benefits achieved with non-blocking extensions over SATA SSDs (on Cluster A). We run both read-only (100% read) and write-heavy (50:50) workloads using a single Memcached client, and a Memcached server configured with 1 GB memory. We pre-load the server with 1.5GB of data, and measure the average Memcached Set/Get latency. From Figure 8(a), we observe that the **H-RDMA-Opt-Block** delivers 54-83% improvement over **H-RDMA-Def-Block**, for both read-only and write-heavy workloads, while **H-RDMA-Opt-NonB-b/i** can achieve 48-80% improvement over both **H-RDMA-Opt-Block** and **H-RDMA-Def-Block**. We observe larger benefits for SATA SSDs as compared to the improvement on NVMe SSDs, due to the higher SSD I/O latency.

G. Evaluation with Bursty I/O Workload Pattern

In this section, we illustrate how the non-blocking API extensions and adaptive I/O schemes can benefit user applications, by modeling the example described in Section IV-B into the micro-benchmarks. We measure the latency involved in reading and writing a block of data to a Memcached cluster made up to 4 servers, with an aggregated memory of 1 GB, using a single client. Each block is broken down into smaller chunks of 256 KB, that are stored in Memcached's 'RAM+SSD' hybrid slabs. We run these experiments with different block sizes, i.e., 2 MB and 16 MB, and an overall workload size of 4 GB, on both NVMe SSDs (Cluster B) and SATA SSDs (Cluster A).

With the non-blocking APIs, the client issues reads or writes for all chunks belonging to a block of data, and then waits for all the chunks to be read from or written into the Memcached server. As opposed to this, the default



(a) Evaluation with Read-only and Write-Heavy Workloads

(b) Evaluation with Bursty I/O Workload Pattern

Figure 8. Evaluation with NVMe SSD

blocking APIs wait for each chunk to be read or written before proceeding to the next. From Figure 8(b), we can see that the **H-RDMA-Opt-NonB-i** scheme can improve the block access latency by 79-85% over **H-RDMA-Opt-Block**, when running over both NVMe and SATA SSDs. We also observe that higher benefits can be obtained for larger block sizes, as there are many more non-blocking operations to be issued; thus more overlapping to exploit.

VII. Conclusion and Future Work

In this paper, we identify and improve the bottlenecks in the current RDMA-enhanced hybrid Memcached design, by proposing new non-blocking extensions to the existing Libmemcached APIs. The non-blocking APIs are designed to provide a quick and easy way for applications to overlap the client-side waits and the server-side SSD access times with other computations or communications. The non-blocking extensions can co-exist with the traditional blocking APIs, while significantly reducing the overall data access latency. We also suggest adaptive I/O enhancements for the Memcached slab manager, in order to achieve high performance with hybrid Memcached running over SATA/NVMe-SSDs.

Through our extensive performance studies, we find that our proposed non-blocking APIs can improve the average Memcached Set/Get latency by up to 16x over the current hybrid design for RDMA-based Memcached, when all data cannot fit in-memory; and up to 3.6x over default Memcached over IPoIB, when all data fits in memory. In addition to bridging the performance gap between the hybrid and pure in-memory RDMA designs, we also improve the server throughput of the hybrid Memcached design by about 2.5x. For future work, we plan on exploring the benefits of employing asynchronous SSD I/O, and the potential of leveraging non-blocking APIs in real-world applications. We also plan to make these designs available through public releases, as a part of the HiBD project [16].

References

[1] libmemcached: Open Source C/C++ Client Library and Tools for Memcached. http://libmemcached.org/.

[2] McDipper: A key-value cache for Flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920.

[3] Memcached: High-Performance, Distributed Memory Object Caching System. http://memcached.org/.

[4] SDSC Comet. http://www.sdsc.edu/services/hpc/hpc_systems.html.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *The Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, Indiana, June 2010.

[7] fatcache. https://engineering.twitter.com/opensource/projects/fatcache.

[8] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004:5–, August 2004.

[9] N. Islam, D. Shankar, X. Lu, M. W. Rahman, and D. K. Panda. Accelerating I/O Performance of Big Data Analytics on HPC Clusters through RDMA-based Key-Value Store. In *International Conference on Parallel Processing (ICPP)*, Sept 2015.

[10] 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 *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, Washington, DC, USA, 2011.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceeding of SIGCOMM 14*, Aug 2014.

[12] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, Cascais, Portugal, October 2011.

[13] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI 14)*, April 2014.

[14] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. CaSSanDra: An SSD boosted key-value store. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 1162–1167, March 2014.

[15] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceeding of USENIX Annual Technical Conference (USENIX ATC 13)*, Jun 2013.

[16] OSU NBC Lab. High-Performance Big Data (HiBD). http://hibd.cse.ohio-state.edu.

[17] 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 *Proceeding of the 41st International Conference on Parallel Processing (ICPP '12)*, pages 470–479, Sept 2012.

[18] D. Shankar, X. Lu, J. Jose, M. Wasi-ur Rahman, N. Islam, and D. Panda. Can RDMA benefit Online Data Processing Workloads on Memcached and MySQL? In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 159–160, March 2015.

[19] D. Shankar, X. Lu, M. W. Rahman, N. Islam, and D. K. Panda. Benchmarking Key-Value Stores on High-Performance Storage and Interconnects for Web-Scale Workloads. In *2015 IEEE International Conference on Big Data (Short Paper)*, Santa Clara, CA, October 2015.

[20] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. BurstMem: A High-Performance Burst Buffer System for Scientific Applications. In *2014 IEEE International Conference on Big Data*, October 2014.

[21] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Characterizing Facebook's Memcached Workload. *IEEE Internet Computing*, 18(2):41–49, 2014.

[22] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Accelerating MapReduce with Distributed Memory Cache. In *15th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 472–478, Dec 2009.