hKVS: a Framework for Designing a High Throughput Heterogeneous Key-Value Store with SmartNIC and RDMA

Hung-Hsin Chen
National Taiwan University
Taipei, Taiwan
r09922038@csie.ntu.edu.tw

Chih-Hao Chang
National Taiwan University
Taipei, Taiwan
d02944024@csie.ntu.edu.tw

Shih-Hao Hung
National Taiwan University
Taipei, Taiwan
hungsh@csie.ntu.edu.tw

ABSTRACT

In-memory key-value store (KVS) is a crucial component of data center applications. Since DRAM provides high bandwidth and low latency, the major performance bottleneck of common in-memory KVS lies in the network stack. Prior works have attempted to replace the traditional network stack with remote direct memory access (RDMA), which achieve orders of magnitude higher throughput and reduce the response latency. To further increase the throughput of an in-memory KVS, we propose a framework called hKVS, which enables the developers to design high-throughput heterogeneous KVS systems by adding the latest generations of smart network interface cards (SmartNIC), such as the NVIDIA BlueField DPU, to the host machines. The hKVS enables a host server to efficiently exploit the computational resources and utilize the RDMA capability of the SmartNICs to offload the workload for the CPU and increase the network bandwidth. The hKVS allows popular key-value objects to be replicated from the host to SmartNIC to form a high-throughput RDMA KVS jointly. We design the architecture of the hKVS, optimize its software implementation, and conduct a series of experiments to evaluate the resulted performance in realistic applications. By adding a SmartNIC to the host, hKVS achieves up to 1.86× and 1.48× higher throughput in 100% and 95% read workloads, which is cost-effective and scalable compared to building a KVS with multiple hosts, considering the SmartNIC costs much less than a high-performance server and multiple SmartNICs can be added to scale the throughput if needed.

CCS CONCEPTS

• **Information systems** → **Information storage systems**; • **Networks** → **Programmable networks**.

KEYWORDS

key-value store, Remote Direct Memory Access (RDMA), SmartNIC, DPU, heterogeneous system

ACM Reference Format:
Hung-Hsin Chen, Chih-Hao Chang, and Shih-Hao Hung. 2022. hKVS: a Framework for Designing a High Throughput Heterogeneous Key-Value Store with SmartNIC and RDMA. In *International Conference on Research in Adaptive and Convergent Systems (RACS '22), October 3–6, 2022, , .* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3538641.3561495

RACS '22, October 3–6, 2022, Virtual Event, Japan

ACM ISBN 978-1-4503-9398-0/22/10…$15.00
https://doi.org/10.1145/3538641.3561495

1 INTRODUCTION

Distributed in-memory key-value stores (KVS) are an essential building block of applications and infrastructures in data centers to share data across multiple nodes. Since these systems use DRAM to store data, they perform better than persistent KVSes and are often used to cache data for fast access. Workload analyses of production KVSes [2, 21] show that a large fraction of requests are read, and most of these read requests correspond to a small fraction of keys.

Although storing the data in DRAM reduces the request handling time, common in-memory KVSes still run on top of the kernel network stack, which becomes the performance bottleneck of in-memory KVSes. Technologies like RDMA provide user-space networking and hardware accelerations to reduce kernel network stack overhead. Therefore, previous studies [4, 8, 9, 20] have used RDMA to replace the traditional network stack for better performance. Based on the usage of server-side CPUs, these RDMA KVSes can be categorized as server-driven KVS, client-driven KVS, or a mixture of them [4]. Like traditional KVSes, server-driven KVSes adopt a request-reply model but replace the socket-based message transmission with RDMA verbs. In contrast, operations in client-driven KVS use one-sided RDMA verbs only.

To further increase the throughput of an in-memory KVS, we propose a heterogeneous KVS architecture which employs a smart network interface card (SmartNIC) to support RDMA and offload the workload for the CPU. SmartNIC, or data processing unit (DPU), is a programmable NIC that allows developers to offload application-specific logic to the NIC. Because of their flexibility and cost-efficiency, SmartNICs have been used in various fields [5, 10, 11, 13, 18]. For example, the NVIDIA BlueField-2 DPU (BF2), a popular SmartNIC device in the market, may deliver similar performance to a high-end x86-based server for client-driven KVSes when the client accesses the server memory using *one-sided* RDMA verbs, which does not invoke the server CPU in handling requests. However, for the server-driven KVSes, the server CPU is required to handle the operations, e.g., querying a hash table, and a high-end x86-based server can significantly outperform the weaker processor on the BF2. Thus, if designed properly, the server may expand the throughput by adding BF2 to the system, but the service latency would not be reduced since the BF2 is not closer to the client than the host. We conduct microbenchmarks on the BF2 to characterize the performance of RDMA verbs and RDMA KVSes on SmartNICs, so that we can properly design a heterogeneous KVS that efficiently uses the resources of the host and SmartNIC.

This paper proposes *hKVS*, a framework to realize a heterogeneous RDMA KVS that runs on the host and SmartNIC. hKVS consists of two KVS services: a primary KVS service (PKVS) on the host and a secondary KVS service (SKVS) on the SmartNIC. The PKVS is built on a server-driven KVS, and the SKVS is built on a client-driven KVS. To offload a portion of read requests to the SKVS, we replicate the popular key-value objects in the PKVS to the SKVS. These popular key-value objects are approximately identified with moving-window access counters. With the offloading mechanism, hKVS can achieve higher overall throughput than a standalone RDMA KVS on the host.

We have implemented the proposed design with the BF2. Experimental results show that hKVS achieves up to 1.86× and 1.48× higher throughput by adding one SmartNIC under 100% and 95% read workloads. At the same time, the replication overhead adds up to 1 µs and 0.3 µs to read and write latency, respectively, under read-intensive workloads. Compared to adding another host to improve the throughput of a KVS, hKVS is very cost-efficient, as the price of a SmartNIC is much lower than a high-end server machine.

The design space of the hybrid KVS spans multiple dimensions, including the implementation of the PKVS and the SKVS, the offload strategy, and the coherence mechanism. In order to deliver optimal performance in the targeted workloads, it is crucial to make careful and proper choices in these fields. The hKVS proposed in this paper targets read-intensive workloads with skewed access. Future works may explore the design space and discover the best configuration for different workloads.

2 BACKGROUND AND RELATED WORKS

This section provides background knowledge on key-value store (KVS) workloads, remote direct memory access (RDMA), RDMA-based KVS, and SmartNIC.

2.1 Key-Value Store Workloads

Previous studies [2, 21] have found several properties in KVS workloads. First, small key-value objects dominate the KVSes in appearance counts and overall weight. Second, most workloads follow a skewed Zipfian popularity distribution [3], in which most of the requests operate on a tiny portion of all key-value objects. Finally, the proportion of each type of request has a strong relationship with the use cases of KVSes. Despite the variety of KVS use cases, most performance-critical KVSes are used in read-intensive scenarios. Therefore, we focus on designing a KVS targeting read-intensive workloads and skewed Zipfian popularity distributions.

2.2 RDMA-based Key-Value Stores

RDMA allows a node to access the memory on remote nodes without involving remote CPUs. With RDMA-capable NICs [16], RDMA provides several performance benefits, including zero-copy, kernel bypass, and low CPU overhead. RDMA [17] supports two types of operations: one-sided verbs and two-sided verbs. When data is transferred with two-sided verbs (`Send` and `Recv`), the CPUs on both sides are involved. In contrast, one-sided verbs (`Read`, `Write`, and atomic verbs) allow a client to operate on remote memory without notifying remote CPUs.

Implementing a KVS with RDMA has been widely studied in previous works [4, 8, 9, 20]. Based on the usage of server-side CPUs, these systems can be categorized as server-driven KVS, client-driven KVS, or a mixture of them [4]. In server-driven KVS, the CPUs on the server are involved in request handling. For example, HERD [8, 9] is a server-driven KVS that adopts a request-reply model, where clients send all KVS requests to the request buffer at the server with `RDMA Write`. The worker threads on the server poll the request buffer, then process the requests by querying or modifying the internal key-value store data structure and reply to clients with `RDMA Send`. A server-driven KVS usually delivers similar performance for either read or write operations since the handling procedures for both operations are symmetric. In addition, it usually delivers higher throughput and lower latency than a client-driven KVS on a high-end server.

In contrast, a client-driven KVS, such as Nessie [4] and Clover [20], executes all KVS operations on the clients with one-sided verbs, so the server-side CPUs are not involved in request handling. The read operations are usually faster than the write operations in a client-driven KVS since a read request can be achieved by one or a few `RDMA Reads`, whereas a write request may involve multiple `RDMA Writes` and RDMA atomic verbs. To strike a balance between server-driven KVS and client-driven KVS, FaRM [7] adopts a hybrid approach, where read is implemented with one-sided `RDMA Read`, and write is implemented with a request-reply model.

2.3 SmartNIC

A smart network interface card (SmartNIC), or data processing unit (DPU), is a programmable NIC that allows users to offload application-specific logic onto the NIC to save the compute resources on the host. Besides high-performance network interfaces, modern SmartNICs usually incorporate general-purpose processors and/or special-purpose accelerators for packet processing, cryptography, and network functions [14]. Because of these benefits, SmartNICs have been used in many fields, including data stores [10, 13] and other applications [5, 11, 18]. For the aforementioned applications, SmartNIC not only improves the performance but also serves as a cost-efficient solution as opposed to adding more servers.

Instead of simply offloading computing services to the SmartNIC, the proposed hKVS combines two independent KVS services on the host and the SmartNIC to build a unified KVS with high throughput. We implement hKVS with an NVIDIA BlueField-2 DPU [14], which is capable of running a complete in-memory KVS service on Linux on its CPU to complement the KVS service on the host.

3 RUNNING KVS ON THE BLUEFIELD-2 DPU

As mentioned in Section 2, it is possible to execute a complete in-memory KVS on the BlueField-2 DPU (BF2) to complement the KVS on the host. Therefore, it is important to characterize the performance of RDMA KVSes on a BF2 to design the proposed hKVS. In this section, we run two types of KVS, Clover and HERD, on the BF2 and observe how the RDMA operations are handled by the BF2. The following subsections describe the experimental setup, evaluate the performance of basic RDMA verbs, and measure the throughput of KVS on the BF2.

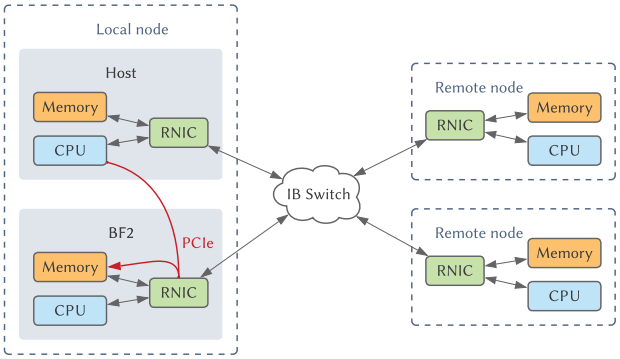

Figure 1: Hardware setup in our experiments.

	AMD Server/Client	Intel Clients	BlueField-2 DPU
CPU	AMD EPYC 7702P	2× Intel Xeon Gold 6148	Arm-v8 A72
#Cores	64	20	8
Nodes	2	2	-
Memory	256 GiB DDR4	384 GiB DDR4	8 GiB DDR4
RDMA NIC	Mellanox ConnectX-6 VPI 100 Gb/s	Mellanox ConnectX-6 VPI 100 Gb/s	Mellanox ConnectX-6 Dx 100 Gb/s
Interconnect	100 Gb/s InfiniBand		

Table 1: Hardware specifications in the experiments.

3.1 Experimental Setup for Evaluating the BF2

Figure 1 illustrates the hardware setup we use to characterize the performance of KVS on the BF2. The server consists of a host machine powered by a 64-core CPU and a BF2 DPU with 8 ARM cores. Both the host and the BF2 DPU are connected to an InfiniBand switch to communicate with remote nodes using RDMA. The detailed specifications are listed in Table 1. Note that the server host can communicate with the BF2 via the PCIe and the RDMA module in ConnectX-6 Dx NIC [15] to transfer data through a dedicated channel with guaranteed bandwidth.

3.2 Latency and Throughput with RDMA Verbs

To characterize the RDMA verbs performance of the BF2, we adopt the approaches in perftest [1] and HERD [8] to measure the throughput and latency of `RDMA Read`, `RDMA Write`, and ECHO. Experiments show that the BF2 has comparable throughput and latency as a high-end server when it is accessed remotely from the clients with one-sided verbs. However, when the BF2 is the issuer of the one-sided verbs, we observed a 0.7 µs increase in `Read` and `Write` latency and a 58% drop in ECHO throughput compared to a high-end server since the less performant CPU on the BF2 is involved in these operations.

3.3 Throughput of RDMA Key-Value Stores

We further analyze the throughput of the server-driven and the client-driven RDMA KVSes on the BF2 with HERD and Clover, respectively. We run the KVS servers of HERD and Clover on the BF2 and populate them with 100K key-value pairs with 8-byte keys before starting the tests. We use 32-byte and 1000-byte values to observe the throughput of the KVS server in workloads with small and large values, respectively. For both RDMA KVSes, we use 16 client threads in each of the three client nodes to deliver the best throughput despite having more than 16 cores on each client node.

Value	32 byte			1000 byte		
Read (%)	100	95	50	100	95	50
HERD	15.5	14.7	14.4	9.6	8.2	4.5
Clover	31.6	14.9	2.5	10.1	8.3	2.0

Table 2: Throughput ($\times 10^6$ op/s) of HERD and Clover on the BF2 DPU under workloads with different read-write ratios.

We adopt the settings of the skewed workloads in the YCSB Benchmark [6] to generate the workload, where the access patterns follow Zipfian distribution with parameter 0.99. Moreover, we choose three read-write ratios (100-0, 95-5, and 50-50) to represent different scenarios in data centers. Table 2 shows the throughput of the KVS servers of HERD and Clover on the BF2 in workloads with 32-byte and 1000-byte values and different read-write ratios.

Experiment results with 32 bytes show that Clover has higher throughput in read-intensive workloads, while HERD performs better in write-intensive scenarios. With 32-byte values, Clover achieves the highest throughput (31.6M op/s) in read-only workload. The throughput drops significantly when the fraction of write operations in the workload increases. On the other hand, the throughput of HERD is bounded by the CPU performance regardless of the read-write ratios since both types of requests in HERD are handled by the CPU on the BF2.

With 1000-byte values, while HERD still performs better in write-intensive scenarios, the advantage of Clover in read-intensive workloads is relatively small. With a large payload, the data transfer time dominates the KVS request handling, and the impact of CPU performance on the throughput is less significant. In fact, in the 100-0 case, both Clover and HERD are bounded by the physical bandwidth limit of the 100Gbps network.

In summary, the experimental results suggest that using a client-driven RDMA KVS delivers better performance on the BF2 than a server-driven RDMA KVS for read-intensive workloads, as the client-driven RDMA KVS consumes fewer CPU resources on the BF2. Based on the observation, we design a hybrid KVS to utilize the strength of the BF2 and increase the overall read throughput by deploying a client-driven KVS on the BF2 to offload read requests when the primary KVS service on the host becomes busy.

4 DESIGN OF hKVS

In this section, we design a heterogeneous RDMA KVS called *hKVS*, which consists of multiple KVS services on the host and the BF2 to deliver higher throughput than a standalone KVS on the host.

4.1 Design Overview

Figure 2 shows the major components of an hKVS running on a machine with a BF2 DPU (the "local node" in Figure 1). The KVS server consists of the primary KVS service (*PKVS*) on the host and the secondary KVS service (*SKVS*) on the BF2. The PKVS runs a

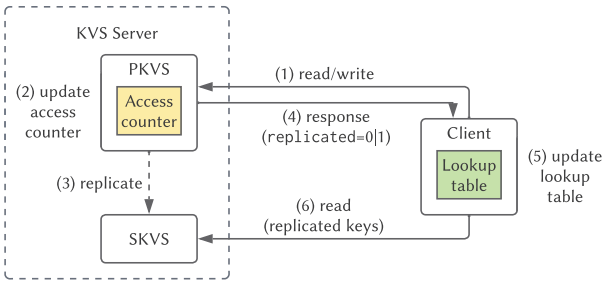

Figure 2: The major components in hKVS.

server-driven RDMA KVS and serves both read and write requests of all keys, while the SKVS runs a client-driven RDMA KVS and serves only the read requests of the popular keys. The PKVS offloads a large portion of the read requests to the SKVS by replicating the keys that are most likely to be read in the future, which comprise only a tiny fraction of all keys. The PKVS uses *moving-window access counters* (Section 4.5) to identify the popular keys when processing requests. The PKVS then replicates the popular keys to the SKVS and synchronizes the changes to the SKVS when the values of the popular keys are updated.

At first, a client sends all its requests to the PKVS. When processing requests, the PKVS updates the access counters and checks if the requested keys have been replicated to the SKVS. The PKVS then sets the `replicated` flag in the response if the requested key has been replicated to the SKVS. The client keeps a lookup table to record the keys available in the SKVS. The client inserts the corresponding key into the lookup table when receiving a response with the `replicated` flag set. After that, the client may send future read requests of these replicated keys to the SKVS, reducing the load of the PKVS and achieving higher throughput.

4.2 Primary KVS Service

The primary KVS service is the KVS service that runs on the host and contains the latest key-value pairs to serve read and write requests from the clients via its independent RDMA NIC. In the proposed hKVS, the PKVS may offload the read requests to the SKVS, but the PKVS is responsible for all write requests. Thus, we build the PKVS based on HERD as it performs well in write-intensive workloads. By offloading the read requests to the SKVS on the BF2, we can save CPU resources on the host and increase the read throughput with the design of hKVS.

Figure 3 shows the components of the PKVS. The PKVS consists of the service threads and the synchronization (sync) threads, and they communicate with each other with the synchronization queue and the completion queue. The service threads handle client requests, and the sync threads synchronize the latest value of the popular keys to the SKVS. To update the value of the popular keys in the SKVS, the service threads send *sync requests* to the sync threads via the synchronization queue and receive completion notifications from the completion queue.

The service threads are similar to the worker threads in the HERD server. Each service thread serves a partition of the key space and stores the key-value pairs of its partition in a local KVS, MICA [12], which HERD uses to store the keys and the mapped addresses of the values. The partition to which a key belongs can be computed

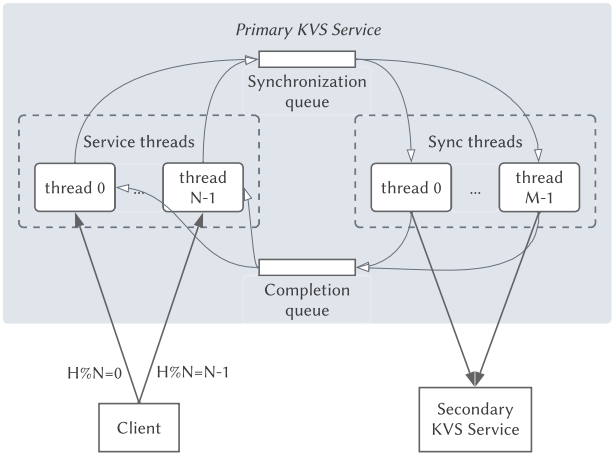

Figure 3: The components in the primary KVS service.

by H mod N, where H is the hashed key and N is the number of service threads. Since each key appears in exactly one local KVS, there is no need to synchronize the key-value pairs between the service threads. To communicate with clients, each service thread creates a request buffer for each client to receive requests from clients. To send a request, clients first compute the target partition of the key, then write the request to the request buffer in the target service thread using `RDMA Write`. Service threads poll the request buffers for each client to check incoming requests, operate on the local KVS, and send the results to clients using `RDMA Send`. In addition, each service thread maintains a set of popular keys for its partition. Whenever a key becomes a popular key, or the value of a popular key is updated, the service thread replicates or updates the key-value pair to the SKVS. Both replication and update are accomplished by enqueuing sync requests to the synchronization queue and polling the completion queue to check the progress of the sync requests. Moreover, the PKVS sets the `replicated` flag in the response when the key in a request has been replicated to the SKVS. Section 4.5 details the method of selecting the popular keys.

Sync threads are responsible for synchronizing the value of the popular keys in the SKVS. They grab sync requests from the synchronization queue, send the new value in the sync requests to the SKVS, and notify the service threads of completion by enqueuing notifications to the completion queue.

When receiving a response from the PKVS with the `replicated` flag set, the client records the key of the corresponding request in a lookup table. To increase the overall read throughput, when the PKVS is busy serving the previous requests from the client, the client can send the read requests of the replicated keys to the SKVS instead of waiting for the responses from the PKVS. In our design, as opposed to actively broadcasting new replicated keys, the client maintains a lookup table to decide if the request should be sent to the SKVS, which scales better as the number of clients grows.

4.3 Secondary KVS Service

The secondary KVS service (SKVS) is the KVS service that runs on the BF2 DPU, acting as a supplementary KVS service which serves the read requests for the popular keys to increase the overall

throughput. We use Clover as the SKVS, as the experiments in Section 3.3 show that Clover delivers higher throughput than HERD in read-intensive workloads on the BF2. Details on Clover are omitted here due to the page limit.

In the proposed hKVS design, the SKVS allows the clients to perform read operations only, but the PKVS can send insert and update requests to the SKVS. This design provides two benefits. First, the chance of failure in `RDMA compare-and-swap` (used in updates) significantly drops since there is at most one instance updating a tail entry simultaneously. Second, since the value of a key is only changed by exactly one service thread in the PKVS, our design eliminates the potential conflicts caused by concurrent writes to the PKVS and the SKVS. Otherwise, a transaction layer is required to ensure consistency between the PKVS and the SKVS.

Finally, we only replicate popular keys to the SKVS since the BF2 memory is small. As discussed in Section 2.1, the requests corresponding to popular keys constitute a large portion of all KVS requests and are enough to utilize the resource on the BF2 properly.

4.4 Consistency between Two KVS Services

To maintain strict or sequential consistency [19] between the PKVS and the SKVS, the service threads in the PKVS must wait for completion notifications before responding to clients. Otherwise, clients may receive outdated values from the SKVS in subsequent read requests before the updates have been synchronized to the SKVS. However, the throughput of the PKVS would be limited by the SKVS synchronization, as the SKVS has a significantly lower write throughput than the PKVS. Therefore, to improve the performance of the PKVS, we update the values of the replicated keys in the SKVS asynchronously by adopting *eventual consistency* [19]. In use cases such as the Twemcache clusters at Twitter [21], the data are updated with best effort, which implies that having stale entries in a KVS is acceptable in these scenarios. With the *asynchronous update*, the service threads no longer need to wait for completion notifications before replying to clients, which improves the throughput and greatly reduces the latency of write requests in the PKVS.

Although asynchronous update improves the throughput, we still limit the number of outstanding sync requests of each service thread to prevent the synchronization queue from piling up. When reaching the limit, the service thread must wait for previous sync requests to complete. Without this limitation, the synchronization queue may consume too much memory and cause out-of-memory errors since the server-driven KVS in the PKVS has a much higher write request throughput than the client-driven KVS in the SKVS. This limitation also helps to alleviate the inconsistency between the PKVS and the SKVS caused by asynchronous updates.

Depending on the consistency and performance requirements for the target use cases, the clients may access hKVS in different modes, including:

- *Extended mode* is the default access mode described in Section 4.2, where clients send requests to the SKVS only when the PKVS is busy. Under this mode, most of the requests are served by the PKVS to deliver lower average latency. However, since clients may read the key-value pairs from the SKVS when the PKVS is busy, the values received from the SKVS might be older than those provided by the PKVS. Thus, this mode does not guarantee consistency, and if the use case requires a stricter consistency model, the interleaved mode mentioned below is preferred.
- *Interleaved mode* can be used when reading a large amount of independent data, e.g., scanning a table, where the client may concurrently read the data from the PKVS and the SKVS to increase its throughput, but the average access latency may be higher since the SKVS resides on weaker hardware. In practice, the client can automatically or manually balance the amount of data to read from the PKVS and the SKVS based on the ratio of their maximum throughputs by configuring the data source of each key to either the PKVS or the SKVS. Note that the write operations are still handled by the PKVS, which enforces the sequential ordering of the write operations. Once the PKVS completes a write operation, the latest value is synchronized to the SKVS. Thus, the aforementioned consistency issue in the extended mode does not exist in this mode.
- *Service level mode* can be used when hKVS is serving multiple clients with different service level objectives (SLO). The hKVS may choose the PKVS to serve the read requests for the clients that require lower latency or stricter consistency. For clients with less SLO, the read requests can be served by the SKVS.

Each application may configure its access mode to meet its requirements. The case study in Section 5 adopts the extended mode to increase throughput for read-intensive applications.

4.5 Determining the Popular Keys

To find the popular keys, we need to analyze the access pattern using an access counter at the PKVS. In the proposed design, each service thread maintains a set of popular keys, so that it can direct its clients to the SKVS for reading the associated values. As the algorithm to maximize the usage of popular key-value pairs really depends on the workload, we provide a mechanism allowing the user to specify a combined policy to determine the popularity of a key-value pair based on how recent and how frequent it has been requested.

For example, the access patterns in Facebook's KVS workloads [2] show high temporal locality and rapid decay in interest in most keys. Recently used keys will likely be reused shortly, and the keys used earlier will not be used in upcoming requests. In this case, the *least recently used* (LRU) eviction policy can be used to determine the key to be evicted (the *victim*) when the number of popular keys reaches a threshold, ensuring these popular key-value pairs are recently used. A key will be replicated to the SKVS if it exists in the set. When the service thread receives a read request from a client, it checks whether the requested key is a popular key. If it is, the service thread updates its access time. Otherwise, it inserts the key into the popular key set and replicates the key-value pair to the SKVS. If a victim is evicted from the set during insertion, it must be invalidated in the SKVS. Invalidation is accomplished with another sync request to update the value of the victim in the SKVS with a special *invalid value* and remove it from the index in the SKVS. Once evicted, future updates on the victim will not be synchronized to the SKVS until it becomes a popular key again. Clients should

remove the key from their lookup tables if the response from the SKVS is identical to the *invalid value*.

Next, we adopt an admission policy that uses a *moving window* to check the access frequency of key-value pairs before adding the key to popular keys since the LRU mechanism does not consider the access frequency of key-value pairs and suffers from frequent replication and invalidation. A key may become a popular key and be replicated to the SKVS only if it has been read at least K times in the last N requests. With this constraint, keys that are only read once in a long period will not be replicated to the SKVS.

Since each time a key is added to or evicted from the popular keys, a sync request is required to replicate or invalidate the key, which incurs a significant amount of overhead if we simply use LRU or frequency alone to determine popularity. With our combined policy, we restrict the replication to the key-value pairs which are both requested recently and frequently, so that the improvement in the overall throughput would not be offset by the replication overhead of these false popular keys. Even though the strict replication policy limits the fraction of read requests that can be offloaded to secondary KVS, it is critical in reducing the overhead of replicating and invalidating unpopular keys. We evaluate the effectiveness and the memory overhead of this approach in Section 5.

In summary, the user may fine-tune the policy to predict which key-value pairs are more likely to be reused in the near future. The combined policy described in this section may be used to provide good predictions for skewed data access patterns, which are often seen in data center KVS operated by social network companies such as Facebook and Twitter. For applications with different data access patterns, the developers may tweak the settings of the combined policy or plug in their custom policies to meet their requirements with the API provided in the hKVS.

5 EVALUATION

In this section, we evaluate the performance of hKVS and compare the performance of the PKVS and the SKVS to HERD and Clover.

5.1 Experimental Setup

The hardware setup for performance evaluation is identical to the setup described in Section 3. In terms of software, the PKVS runs on the 64-core AMD server, and the SKVS runs on the BlueField-2 DPU. We use one AMD client and two Intel clients as the KVS clients with 24 threads on each client unless otherwise stated, since further increasing the number of client threads does not increase the throughput. Each client thread can issue 64 outstanding requests to increase the concurrency. The hKVS is configured as the following for performance evaluation:

- Each service thread in the PKVS is allowed to have at most 128 outstanding sync requests at the same time to prevent out-of-memory errors.
- The number of popular keys is limited to 100,000, which is sufficient for the SKVS to offload roughly 72% key-value requests from the PKVS in our experiments with 8M keys under the Zipfian distribution.
- The numbers of service threads and sync threads in the PKVS are varied during the scalability tests conducted in Section 5.2. For the other tests, the server uses 32 service threads and 24 sync threads.
- We configure the admission policy mentioned in Section 4.5 with $K = 4$ and $N = 800000$, as this particular setting reduces the eviction rate to 0 and delivers the highest throughput. With this setting, the data structures for recording frequency and recency information consume 15MB of memory.

We carry out two types of tests to measure the performance of a KVS. While each key-value pair is indexed by an 8-byte key, the values for all keys in the short-value test are 32-byte long, whose performance would be mainly affected by the transaction processing overhead. The values in the long-value test are 1000-byte long, which should produce higher data throughput compared to the short-value test. During the initialization, we populate the PKVS with 8M key-value pairs for the short-value test and 260K for the long-value test, respectively, which occupies approximately 256 MB in the host memory. Similar to the settings in the YCSB benchmark, the access pattern of the key-value pairs in our tests follows the Zipfian distribution (parameter 0.99). Before we measure the performance, we have each client warm up the set of popular keys in each service thread of the PKVS with 10,000 read requests.

5.2 Thread Allocation in the PKVS

Since the PKVS comprises multiple service threads and sync threads, the thread allocation for each type of worker thread is crucial in determining the performance of the PKVS. To find the best thread allocation, we measure the throughput by varying the numbers of service threads and sync threads in the PKVS.

5.2.1 The Number of Sync Threads. First, we measure the throughput of the PKVS with 24 service threads and vary the number of sync threads. As the major use of the sync threads is to update the value in the SKVS, we measure the throughput of the PKVS with write-only workloads using different numbers of sync threads.

Figure 4 shows the throughput of the PKVS using 32-byte and 1000-byte values with different numbers of sync threads. The throughput of the PKVS scales linearly up to 24 threads and is then bounded by the write throughput of the SKVS. Although using more threads still results in slightly improved throughput, it also increases the power consumption, which is why we decided to use 24 sync threads in the following experiments in this section to reserve CPU resources and save the power consumption for the service threads.

5.2.2 The Number of Service Threads. Now that we have decided the number of sync threads, we measure the throughput of the PKVS with different numbers of service threads under workloads with different read-write ratios. We use 32-byte values to emphasize the cost of request handling instead of data movement.

Figure 5 shows the throughput of the PKVS with different numbers of service threads. With 100% read requests, the throughput of the PKVS (the blue line) scales well initially but then slowly peaked at 66M op/s with 32 threads. Upon further analysis, the performance is limited by both load imbalance and lock contention on the synchronization queue. When write requests occur, the PKVS needs to synchronize the SKVS if the key is popular, and the throughput of the PKVS is limited by the write throughput of the SKVS, which is 3.7M op/s with 24 sync threads, according to Figure 4. Based on

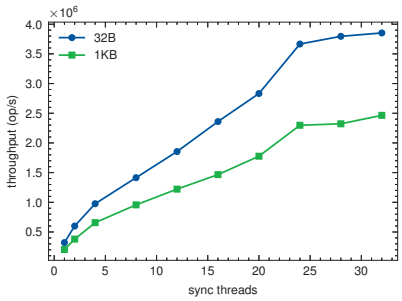

Figure 4: Throughput of the PKVS in write-only workloads with different numbers of sync threads.

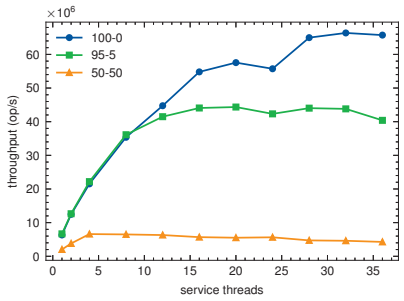

Figure 5: Throughput of the PKVS on workloads with 32-byte values and different read-write ratios.

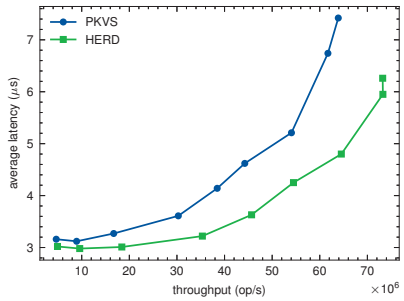

Figure 6: Throughput versus latency of read requests with the PKVS and HERD.

these results, we set the number of service threads to 32 to measure the performance for read-intensive workloads.

5.3 Comparison between PKVS and HERD

In this subsection, we measure the latency and throughput of the PKVS and compare the result to HERD. Since the PKVS is based on HERD, we can evaluate the impact of additional operations, such as popular key selection and update synchronization, on its performance by comparing the throughput and latency between the PKVS and HERD. We use 32-byte values to emphasize the overhead added to the request handling logic.

5.3.1 Overhead of Popular Key Selection. We quantify the overhead of the popular key selection mechanism implemented in the PKVS by changing the number of client threads to vary the load level and compare the service latency and throughput of read requests between the PKVS and HERD. Figure 6 shows the read throughput versus average latency of the PKVS and HERD. For both PKVS and HERD, the latency increases as the throughput increases due to the batching mechanism implemented by HERD. Since a service thread processes the requests from all clients in a batch, the batch size increases as the number of clients increases, resulting in larger differences in average latency under higher throughput. When the throughput is less than 10M op/s, the difference in latency is about 0.16 µs, as the batch size is 1. When the throughput is increased to 54M op/s, the batch size is increased to 6, so the overhead of latency in updating the moving-window access counters and the timestamps for the LRU policy is enlarged 6 times and amounts to 1 µs, roughly 23% of overhead. In terms of maximum throughput, the PKVS delivers 64M op/s while HERD delivers 73M op/s, a 15% drop due to the popular key selection mechanism.

5.3.2 Limited Write Performance. Even though hKVS is designed for read-intensive workloads, we still measure the write performance and report its limitation due to the synchronized updates to the SKVS. Figure 7 shows the latency and throughput of the write requests of the PKVS and HERD. We adjust the load by changing the number of client threads. Under light load, the service threads can enqueue sync requests in a non-blocking way, which adds only 0.3 µs to the average latency. However, the maximum throughput can only reach 3.7M op/s, which drops significantly from HERD

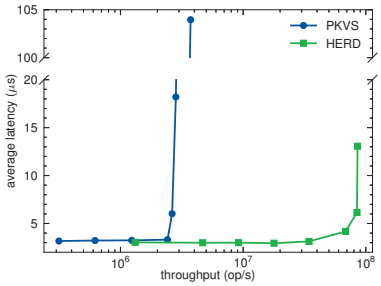

Figure 7: Throughput versus latency of write requests with the PKVS and HERD.

(85.6M op/s). Thus, we do not recommend using hKVS for write-intensive workloads. Fortunately, write requests comprise a tiny fraction (1%∼2%) of all requests in some Twemcache clusters [21].

5.4 Overall Throughput Comparison

In this subsection, we measure the throughput of the proposed hKVS and compare it to an ideal hKVS, which ignores the overhead of synchronization and popular keys selection. As shown in Table 3, the throughput of the actual hKVS is composed of the measured throughput on the PKVS and the SKVS, and the throughput of the ideal hKVS is estimated by calculating the total throughput of HERD and Clover. Note that HERD and Clover are essentially ideal PKVS and SKVS without synchronization and popular keys selection.

When the values are 32-byte long, the throughput is bounded by the CPU and the request rate of the RDMA NICs. In the case of 95% read, the overall throughput for hKVS is 45.2M op/s, which is less than the 74.6M op/s offered by a standalone HERD server, but hKVS achieves 91.5M op/s in the 100% read case, providing 1.26× higher throughput than HERD. Basically, for short values, the impact of synchronization overhead can be very severe, as 5% write operations can result in a 50% drop in the throughput. On the other hand, both SKVS and Clover are also seriously impacted by the 5% write operations since more `RDMA Reads` are required to fetch the latest values when the KV entry address cached at the clients no longer points to the latest entry.

With 1000-byte values, the throughput of hKVS is not impacted as much by the 5% write operations as the throughput is bounded by the network bandwidth (100 Gb/s in our setup), which ideally

Value (byte)	Read (%)	Throughput ($\times 10^6$ op/s)					
		Actual hKVS			Ideal hKVS		
		PKVS	SKVS	Total	HERD	Clover	Total
32	95	32.1	13.1	45.2	74.6	14.9	89.5
	100	64.6	26.9	91.5	72.5	31.6	104.1
1000	95	11.2	7.0	18.2	12.3	8.2	20.5
	100	11.0	10.8	21.8	11.7	10.1	21.8

Table 3: The throughput of the actual and ideal hKVS.

allows 12.5M op/s, and both PKVS and HERD have achieved more than 11M op/s. The addition of SKVS overcomes the bandwidth limit and increases the throughput of hKVS to 18.2M op/s for 95% read and 21.8M op/s for 100% read, which are quite close to the ideal total throughput and are 1.86× and 1.48× higher than a standalone HERD server. Actually, one could add a regular NIC for the PKVS to overcome the bandwidth limit, but eventually the throughput of the PKVS is still bounded by the CPU utilization. With hKVS and SmartNIC, our solution not only increases network bandwidth but also provides more CPU resources and a variety of offload modes.

6 CONCLUSION AND FUTURE WORKS

In this paper, we evaluate how BlueField-2 DPU performs for various RDMA KVSes and propose hKVS, a heterogeneous RDMA KVS, to efficiently utilize the resources of the host and BlueField-2 DPU to increase the throughput for read-intensive workloads. As the BlueField-2 performs well for read operations, we design a mechanism to replicate frequently-read key-value objects to a secondary KVS on the BlueField-2 to increase the read throughput. Experimental results show that the proposed hKVS can achieve up to 1.86× and 1.48× higher throughput than a standalone RDMA KVS in 100% and 95% read workloads by adding a BlueField-2 DPU to the host. As the price of a SmartNIC is much lower than a high-end server, our approach is an attractive solution. Further analysis of the results provides guidelines for building a cost-effective, high-throughput hybrid RDMA KVS with a high-end server and SmartNICs.

We believe the proposed design can be further improved and extended in several ways. First, we may connect multiple SKVSes to the PKVS to scale up the throughput. Moreover, although the proposed hKVS focuses on read-intensive workloads, as discussed in Section 1, one may explore the design space to tweak the architecture of hKVS for specific workloads by making different decisions. For example, to deliver better throughput in write-intensive workloads, one may build the SKVS with an RDMA KVS featuring higher throughput in write requests, such as HERD, to improve the synchronization performance. When the targeted workload has a uniform access distribution, the offload strategy may prioritize small key-value pairs to increase the number of key-value pairs replicated to the SKVS. Hopefully, the approach, evaluation method, and discussions described in this paper will serve as good references for future work.

ACKNOWLEDGMENTS

We thank Fang-An Kuo, Kuo-Teng Ding, and Yi-Lun Pan from National Center for High-performance Computing for providing the experiment platforms and their valuable insights and suggestions.

REFERENCES

[1] 2021. linux-rdma/perftest: Infiniband Verbs Performance Tests. (2021). https://github.com/linux-rdma/perftest Version: 4.5-0.2.
[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. New York, NY, USA, 53–64.
[3] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, Vol. 1. 126–134 vol.1.
[4] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. 2017. Nessie: A Decoupled, Client-Driven Key-Value Store Using RDMA. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3537–3552.
[5] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2020. λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 67–77.
[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. New York, NY, USA, 143–154.
[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA, 401–414.
[8] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. New York, NY, USA, 295–306.
[9] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO, 437–450.
[10] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. New York, NY, USA, 756–771.
[11] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. 2017. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. New York, NY, USA, 506–519.
[12] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA, 429–444.
[13] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. New York, NY, USA, 318–333.
[14] NVIDIA. 2021. NVIDIA BlueField-2 Datasheet. (2021). https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf Accessed: 2022-04-20.
[15] NVIDIA. 2022. Functional Diagram - BlueField DPU OS 3.8.5 - NVIDIA Networking Docs. (Jan. 2022). https://docs.nvidia.com/networking/display/BlueFieldDPUOSv385/Functional+Diagram Accessed: 2022-04-30.
[16] NVIDIA. 2022. NVIDIA InfiniBand Adapters. (2022). https://www.nvidia.com/en-us/networking/infiniband-adapters/ Accessed: 2022-05-02.
[17] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. 2007. A Remote Direct Memory Access Protocol Specification. RFC 5040. (2007).
[18] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. New York, NY, USA, 740–755.
[19] Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed Systems: Principles and Paradigms* (2 ed.). Pearson Prentice Hall, Upper Saddle River, NJ.
[20] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 33–48.
[21] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 191–208.