DPU-KV: On the Benefits of DPU Offloading for In-Memory Key-Value Stores at the Edge

Arjun Kashyap
University of California, Merced
Merced, CA, USA
akashyap5@ucmerced.edu

Yuke Li
University of California, Merced
Merced, CA, USA
yli304@ucmerced.edu

Xiaoyi Lu
University of California, Merced
Merced, CA, USA
xiaoyi.lu@ucmerced.edu

Abstract

In-memory key-value stores (KVS) are widely used for edge data storage, where low latency and high throughput are essential. Data Processing Units (DPUs), with their low power use and offloading capabilities, suit resource-constrained edge computing. While DPUs offer a new design point for KVS, their offloading in edge environments remains underexplored and challenging. In this paper, we unveil the potential of offloading in-memory CPU-based KVS to SoC-based DPUs, specifically NVIDIA's BlueField-2 (BF-2) and BlueField-3 (BF-3), with the aim of enhancing KVS performance. We propose a principled exploration methodology of dividing a KVS (i.e., MICA) into its logical components and identifying the CPU-intensive KVS component (i.e., communication engine). Next, we perform fine-grained offloading analysis and explorations on DPUs. To maximize benefits in terms of latency and throughput from fine-grained KVS offloading on DPUs, we propose a series of significant performance optimizations, including a key-value-based queue-pair model, overlapped KV request/response processing, reduced DMA operations per KV batch, dual-communication engine, and a sharding-based design. Our key finding is that our proposed fine-grained KVS offloading designs on modern DPU architectures (i.e., BF-2 and BF-3) can provide much lower latency (up to 68%) and higher throughput (up to 36%) than MICA (*CPU-only*) and coarse-grained DPU offloading schemes at the edge. To our knowledge, this paper is the first to explore the performance benefits of fine-grained KVS offloading to DPUs at the edge.

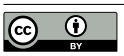
CCS Concepts

• **Hardware** → **Networking hardware**; • **Networks** → *In-network processing*.

Keywords

Data Processing Units (DPUs), Key-Value Stores, Offloading, Edge

ACM Reference Format:
Arjun Kashyap, Yuke Li, and Xiaoyi Lu. 2025. DPU-KV: On the Benefits of DPU Offloading for In-Memory Key-Value Stores at the Edge. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, Notre Dame, IN, USA, 14 pages. https://doi.org/10.1145/3731545.3731571



HPDC '25, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731571

1 Introduction

Motivation: Edge computing entails deploying computational resources near end-users, typically at the network's edge. This paradigm enables low-latency, high-throughput processing, which is essential for a broad range of performance-sensitive applications running at the edge. Prominent examples of such applications include real-time machine vision [65], stream processing [25], web applications [5, 16], autonomous vehicular networks [4, 66], and federated learning [12, 83]. A key enabler for these edge-based workloads is the use of in-memory key-value stores (KVS), which are often deployed entirely at the edge to meet strict performance requirements [15, 52, 55].

On the other hand, hardware accelerator devices have shown significant promise for edge computing [53, 82]. Among them, Data Processing Units (DPUs), also known as SmartNICs, are gaining increasing popularity in edge environments [10, 30, 51, 54, 56] due to their low power consumption and ability to help edge settings to better utilize their limited computing resources. DPUs can help host applications offload certain tasks and improve performance. Unfortunately, offloading KVS tasks to DPU in an edge setting is less explored. We observe that co-designing KVS with DPUs can help enhance KVS performance, as KVS in the edge requires both low latency and high throughput [25, 49]. Additionally, edge servers can be resource-constrained [49, 67, 73], and offloading KVS tasks can help free up resources, enabling resource sharing among other edge services and applications [27, 46, 81].

However, achieving optimal KVS performance when offloading to DPUs is non-trivial. It is crucial to consider the following *challenges* when designing DPU-offloaded KVS in the edge.

❶ Firstly, it is essential to determine which KVS component to offload. Although offloading the entire KVS to DPUs (*DPU-only*) is easier, this coarse-grained offloading approach may not yield the best KVS performance (§ 3.2). Therefore, it is vital to perform a fine-grained decomposition of KVS and its offloading.
❷ Secondly, a fast host-DPU data path is required to enable communication between the KVS component on the host and the DPU. The challenge lies in creating a low-latency data path using the low-level data movement primitives available with DPUs.
❸ Thirdly, it is necessary to mitigate the additional data movement overheads that arise from offloading a KVS component. An effective KV processing scheme between offloaded and non-offloaded KVS components that hides the extra data movement latency is required.
❹ Additionally, a naive allocation of DPU/host resources between offloaded and non-offloaded KVS components could lead to high host resource usage. When low host resource utilization is necessary, an intelligent resource mapping between the (non-)offloaded KVS components is needed.

❺ Finally, the DPU-offloaded KVS should be able to provide low latency and high throughput in comparison to CPU-based KVS.

Limitation of state-of-the-art approaches: Prior works [9, 25, 55, 65, 68, 72] primarily focus on designing KVS for the edge and do not explore co-designing KVS with hardware devices like DPUs. While DPUs are widely utilized in the edge [10, 30, 51, 54, 56], prior works have not investigated their potential advantages for KVS in edge environments. Our work differs by focusing on how to co-design KVS with DPUs in edge environments to increase performance. We defer detailed discussion of related work to § 2.2.

Key insights and contributions: Figure 1 highlights our fine-grained KVS offloading methodology, outlining its challenges, key contributions, and the benefits achieved through DPU offloading. We begin our exploration (**C1**) by logically partitioning the KVS into fine-grained components—namely, the communication and processing engines. To identify the most suitable component for offloading, we profile the KVS system to determine which part is the most CPU-intensive. Our profiling results reveal that the communication engine consumes up to 70% of host CPU cycles (§ 3.3), indicating its potential as an effective candidate for DPU offloading.

After identifying the communication engine as the target for offloading, the next step in our exploration (**C2**) is to establish an efficient data movement path between the offloaded and non-offloaded KVS components. To this end, we propose a key-value-based queue-pair model for host-DPU communication, inspired by well-established high-performance queue-pair programming models [8, 47, 78]. Additionally, we utilize the DPU's low-level DMA primitives to construct a low-latency data path between the host and the DPU.

Achieving high-performance KVS through fine-grained offloading to DPUs is a complex task. Hence, to complete our offloading journey, we introduce a series of performance optimizations for the DPU-offloaded KVS. To further mitigate the data movement overhead introduced by offloading, we overlap the key-value request and response processing stages between the DPU and the host, while also optimizing each stage individually (**C3**). This enables low latency and necessitates a one-to-one host-to-DPU core mapping between the processing and communication engines.

To reduce host resource usage in resource-constrained edge environments, we improve the core mapping between the processing and communication engines across the host and the DPU (**C4**).

We observe that when the DPU's SoC cores are weaker than the edge host CPU cores, the offloaded communication engine alone may underutilize the host-side KVS processing engine. To address this problem and meet the goal of **C5**, we introduce a dual communication engine design (§ 4.5) that distributes the communication workload across both the DPU and a subset of the host CPU cores freed in **C4**, thereby increasing the request injection rate to the host processing engine and improving overall throughput. We also explore an alternative sharding-based design as a comparative strategy. The performance trade-offs of these approaches are evaluated against other offloading architectures in § 6, demonstrating the efficiency and effectiveness of our various *DPU-KV* designs.

The key insight is that fine-grained offloading of the CPU-intensive KVS component to the DPU, combined with our proposed performance optimizations, can boost KVS performance. The major **contributions** of this paper are as follows:

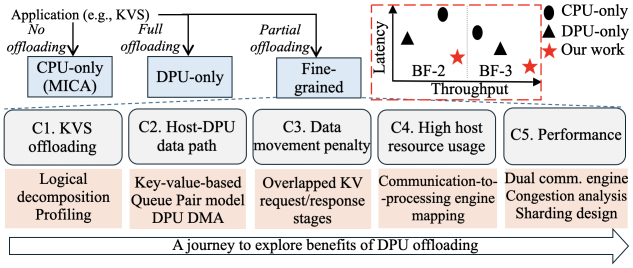


Figure 1: Our proposed exploration methodology, unveiled challenges (C1-C5), contributions, and achieved benefits. ***Our key finding is fine-grained KVS offloading on DPUs (BF-2 & BF-3) can provide lower latency (up to 68%) and higher throughput (up to 36%) than CPU-only and coarse-grained DPU offloading at the edge.***

❶ We propose a principled fine-grained offloading methodology of in-memory KVS to DPUs at the edge.
❷ We present a set of performance optimization strategies to maximize the performance of DPU-offloaded KVS.
❸ Our proposed fine-grained KVS offloading designs (*DPU-KV*) on modern DPU architectures can provide much lower latency (up to 68%) and higher throughput (up to 36%) than CPU-only KVS and coarse-grained DPU offloading schemes at the edge.
❹ To the best of our knowledge, this is **the first work** to investigate the performance advantages of fine-grained KVS offloading to DPUs at the edge.

Experimental methodology: We prototype our work with SoC-based DPUs–NVIDIA's BlueField-2 [59] (BF-2) and the latest generation BlueField-3 [60] (BF-3)—and state-of-the-art CPU-based in-memory KVS, MICA [42, 43]. We evaluate KVS's performance across three YCSB [17] workloads—YCSB A (write-heavy), YCSB B (read-heavy), and YCSB C (read-only). We use both uniform and skewed key distributions for KV request generation, as skew is common in edge environments due to location-based variations in input workload traffic [25]. We show that KVS communication offloading to BF-2 DPU reduces host resource utilization while providing performance. Our design saves 62.5% host cores and reduces latency by up to 68% than MICA (*CPU-only*) KVS. It also achieves up to 1.1× and 1.9× higher throughput over CPU-only and DPU-only KVS, respectively. With BF-3 DPU, offloading the KVS communication engine lowers latency by up to 33% and increases throughput by up to 36% compared to CPU-only KVS.

Limitations of the proposed approach: We assume that a separate service would manage replication and ensure the availability of KVS in case of edge node failure, and leave the exploration of offloading these tasks to DPUs for future work.

2 Background and Related Work

2.1 Background

Choosing In-memory KVS. Popular open-source in-memory KVS include—Memcached [1] and its variants like MemC3 [23], Redis [2], and MICA [42, 43]. Redis is a low-latency KVS widely used in both

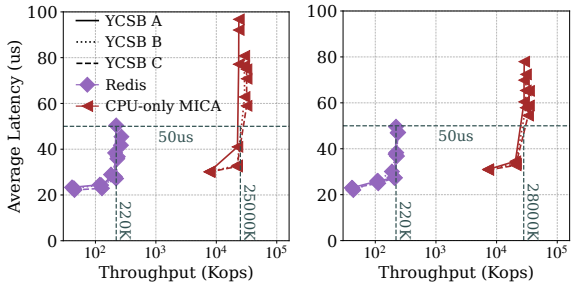


Figure 2: Performance comparison of in-memory Redis and MICA KVS on host. Evaluation with YCSB workloads and two key distributions. Note x-axis is in log scale. At 50μs latency, MICA shows up to 127× throughput than Redis.

edge [9, 55, 63] and cloud [14, 79] environments. MICA is a highly performant KVS [7, 38] and is widely used in RPC applications [31] and many key-value store backends [32, 33, 74, 80]. We find that Redis delivers low latency but suffers from low throughput. In Figure 2, we plot the performance of both Redis (v7.4.1) and MICA KVS for YCSB [17] workloads for uniform and skewed key distributions. The testbed details are described in § 3.1. Both KVSs use 8 server threads, 12 client threads, and the same-sized KV items (8 B key, 8 B value). We observe that Redis has two orders of magnitude lower throughput than MICA at a similar latency. For example, at 50μs latency, Redis achieves 220K ops/sec, while MICA reaches 28,000K ops/sec, resulting in a 127× difference in throughput. The stark performance gap between Redis and MICA is because MICA has been streamlined for performance for typical KVS workloads, efficiently handling both read- and write-heavy patterns as well as different key distributions. Edge datastores are expected to deliver both low latency and high throughput [25, 49]. Hence, we choose MICA to study the performance benefits of DPU offloading for KVS.

Data Processing Units. DPUs are advanced Network Interface Cards (NICs) equipped with programmable compute (FPGA/MIPS/SoC) that allow hosts to offload tasks. Many vendors offer DPU products like NVIDIA BlueField (BF) [57, 59, 60], AMD Pensando [6], Marvell Octeon [50], and Intel IPU [28]. Among them, SoC-based DPUs, e.g., BF DPUs, are easier to program. BlueField DPUs contain low-powered ARM/SoC cores running their own operating system, hardware accelerators (e.g., DMA engine), a host channel adapter (HCA) (e.g., ConnectX-6/7), and on-board DDR memory as shown in Figure 3.

The BlueField DPUs we use include an internal PCIe switch connecting the SoC cores and NIC to the host. This enables the host to utilize the DPU's SoC cores for offloading functions while allowing the DPU/host to access host/DPU memory via the DMA engine. DPU supports three modes—NIC mode, separated host mode, and DPU mode. In NIC mode, the DPU acts as a regular NIC with the ARM subsystem disabled, preventing any offloading to ARM cores. In DPU mode, the embedded ARM subsystem is active and assumes ownership and control of the NIC. In separated host mode, DPU functions as an independent server with its own networking ports, separate from those of the host. The separated host mode allows both the ARM and the host to use and operate network interfaces simultaneously.

In this paper, we use BF-2 in the separated host mode as it provides the most flexibility to explore different DPU-offloaded designs. As BF-3 no longer supports the separated host mode, we use it in the DPU mode. In this mode, all network traffic to the host is routed through a virtual switch managed by the DPU before reaching the host CPU cores [61].

2.2 Related Work

Prior works [9, 25, 55, 65, 68, 72] study how to design KVS at the edge. SDKV [9] and Portkey [55] focused on KV data placement at the edge to reduce request latency. EdgeKV [68] was designed for the network edge to deliver low-latency KV access while providing data privacy. Fogstore [25] was designed to share application states of event-based systems while ensuring consistency and resilience against geographically correlated failures. Ravindran et al. [65] created a key-value edge datastore for performing vision analytics at the edge. CaseDB [72] is a log-structured merge (LSM) tree-based KVS that utilizes SSD for value storage at the edge. Our work is different as it focuses on studying how offloading KVS functions to novel hardware, such as DPUs, in edge environments can help improve performance.

DPUs are becoming increasingly popular at the edge [10, 30, 51, 54, 56]. Barsellotti et al. [10] discussed how DPUs could be utilized for edge telemetry and offloading network security functions. Ni et al. [54] offloaded load balancing tasks of middlebox edge servers to DPUs. Jowett [30] demonstrated edge host CPU reduction by offloading software-defined networking (SDN) network controller to the DPU. While previous works focus on offloading various applications to DPUs at the edge, we specifically examine the benefits of offloading KVS to DPUs in edge environments.

Some studies [35, 37, 45, 71, 76] have conducted performance characterization of BlueField DPUs. Our prior study [35] analyzed three generations of DPUs across network, DMA engine, and on-board memory to identify their idiosyncrasies and provided system/hardware design guidelines. Wei et al. [76] characterized different communication paths of DPU and proposed recommendations for offloading RDMA-based KVS (DrTM-KV [77]) to DPUs. Liu et al. [45] characterized the networking and computing capabilities of DPU. Li et al. [37] provided guidelines for optimizing network middleboxes with DPUs. Thostrup et al. [71] evaluated DPUs for database operations.

Other works have studied offloading KVS tasks to DPUs in cloud [13, 36, 48, 70] and other tasks to DPUs in the HPC setting [11, 29, 39]. KV-direct [36] utilized an FPGA-based SmartNIC/DPU to improve the performance of in-memory KVS. iPipe [48] utilized a MIPS-based DPU to improve the performance of disk-based persistent KVS. SKV [70] offloaded KVS data replication to DPU. iKnowFirst [13] used DPU to assist in the compaction of LSM-tree-based KVS. BluesMPI [11] offloaded MPI non-blocking Alltoall collective operation to overlap communication and computation. DPUs have also been utilized to accelerate workloads like molecular dynamics [34] and compression [39–41].

To the best of our knowledge, this paper is the first work to explore the performance benefits of fine-grained KVS offloading to DPUs at the edge.

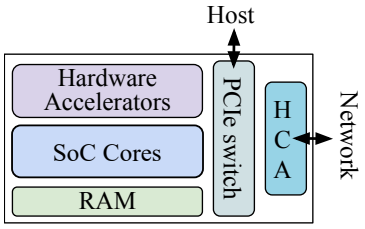


Figure 3: Overview of BF-2/3 DPU architecture.

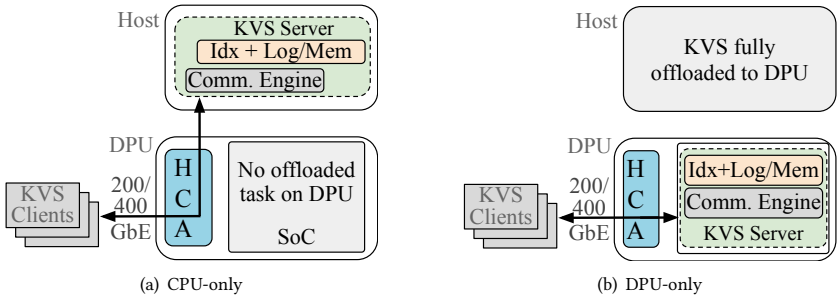


Figure 4: (a) KVS running fully on the host (*CPU-only*), (b) KVS fully offloaded to BF-2/3 DPU (*DPU-only*).

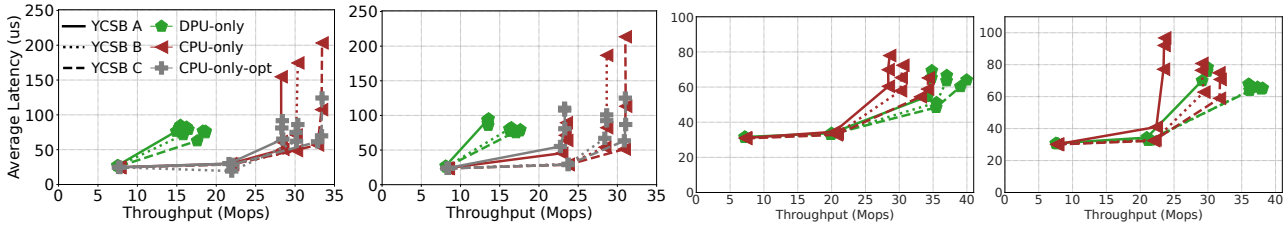


Figure 5: End-to-end latency vs. throughput for CPU-/DPU-only KVS for YCSB A, B, and C for skewed and uniform key distributions with (a)–(b) BF-2; (c)–(d) BF-3. With BF-2, *CPU-only* achieves higher throughput than *DPU-only*, whereas with BF-3, *DPU-only* shows higher performance than *CPU-only*.

3 Methodology

This section presents our principled methodology for fine-grained key-value store (KVS) offloading. We first describe the evaluation platform, followed by an analysis of coarse-grained offloading performance to DPUs. Using BF-2 and BF-3, we compare CPU-only KVS with the fully/coarse-grained DPU-offloaded (*DPU-only*) KVS design. Finally, we profile the CPU-based KVS to identify decomposable components suitable for fine-grained offloading to DPUs.

3.1 Evaluation Platform

Our edge testbed consists of two nodes, one as a server and the other as a client/load generator. Each node is equipped with a 6-core 3.3 GHz Intel Xeon E-2136 CPU (12 cores with Hyper-Threading enabled), 3200 MHz 32 GiB DDR4 memory, PCIe 3.0 x16, and 12 MiB L3 cache, and runs AlmaLinux 8.5 (Linux 4.18.0). We use moderately lower-performance hosts to mimic lower-power edge systems. Each node either contains NVIDIA BlueField-2 [59] or BlueField-3 [60] DPUs. It is challenging to access a testbed that includes both BF-2 and BF-3 with identical supporting components. Our setup allows us to keep the hardware and software configuration consistent, varying only the DPU model based on the experiment.

The BF-2 and BF-3 DPUs are connected back-to-back via InfiniBand HDR (200 Gbps) and NDR (400 Gbps) cables, respectively. The BF-2 consists of 8 cores (ARM Cortex-A72 processor @ 2.5 GHz), 16 GB on-board DDR4 memory, and one 200 Gbps InfiniBand/Ethernet port. BF-3 features 8 cores (ARM Cortex-A78 @ 3 GHz), 16 GB DDR5 memory, and one 400 Gbps InfiniBand/Ethernet port. BF-2 runs Ubuntu 20.04.5 (Linux 5.4.0) and BF-3 runs Ubuntu 22.04 (Linux 5.15). In our testbed, BF-2 and BF-3 utilize NVIDIA DOCA SDK [62] (v1.5.0 and v2.5.0) for host-DPU data movement. Evaluating both BF-2 and BF-3 highlights the impact of different hardware generations, with BF-2 delivering lower performance than our edge host and BF-3 surpassing it.

We run MICA KVS in Exclusive Read Exclusive Write mode, where each key partition is assigned to a single core and use its cache semantics, as the performance difference between MICA store and cache is negligible [43]. We do not modify KVS's memory allocator and indexing schemes. We utilize MICA's key-value request generator to generate 192 M key-value pairs for both uniform and skewed (Zipf with 0.99 skewness) workloads. Evaluating performance under skewed workloads is crucial, as traffic patterns can fluctuate in edge environments [25]. We use the default 8 B key and 8 B value sizes from MICA and vary the GET-to-PUT ratio to generate write-heavy (50% GETs, 50% PUTs), read-heavy (95% GETs, 5% PUTs), and read-only (100% GETs) workloads that correspond to YCSB's [17] A, B, and C workloads, respectively. We report latency at peak throughput similar to [36] under different KV request loads by varying the number of KVS client threads. The client node runs the KVS key-value generator on all 12 cores for 90 seconds. The results reported are an average of three runs.

3.2 Coarse-Grained KVS Offloading to DPU

Here, we compare the performance of CPU-based KVS and coarse-grained offloading of KVS to DPUs when using BF-2 and BF-3.

In-memory KVS server consists of two major components—a processing engine and a communication engine. The processing

engine performs all the key-value-related operations like indexing, maintaining KV data structures, and storing/retrieving a *(key, value)* pair in/from memory. The communication engine handles network I/O to receive/send packets from the KVS clients. Figure 4(a) illustrates the high-level architecture of an in-memory CPU-based KVS, where both the processing and communication engines run on the host, referred to as *CPU-only*. *CPU-only* utilizes the HCA of BF-2 or BF-3 DPU, depending on which DPU is connected to the host, effectively making the DPU function as a NIC to serve client KVS requests. Figure 4(b) illustrates a coarse-grained method to offload the KVS entirely to BF-2/3 DPU, which we refer to as *DPU-only*.

Figure 5(a) and Figure 5(b) show the end-to-end latency vs. throughput for the MICA KVS running completely on CPU (*CPU-only*) or DPU (*DPU-only*) when using BF-2. CPU-only KVS has 1.73×–1.76× higher throughput and 0.95×–2.68× higher latency than *DPU-only* for key-value requests across different workloads. The increased latency of *CPU-only* is due to congestion in the server NIC's receive queue. The KVS client dynamically adjusts its request generation rate to match the server's response rate, ensuring that the KVS server is not starved. However, requests accumulate in the receive queue at peak throughput, causing a significant latency spike. When we doubled the receive descriptors on the server side from 128 to 256, the latency for YCSB C reduced by 41% (shown as *CPU-only-opt* in Figure 5(a)–(b)). For lower client request generation rates, this queuing behavior is absent in the *CPU-only* case as the request processing and generation rates are lower, preventing any backlog in the receiver's NIC. *DPU-only* exhibits comparable latency (76–93μs) at lower peak throughput (13.4–17.6 Mops) due to the wimpier nature of BF-2's SoC cores compared to the host.

Similarly, Figure 5(c) and Figure 5(d) illustrate the performance of *CPU-only* and *DPU-only* when using BF-3. Across different workloads and key distributions, *DPU-only* exhibits superior performance than *CPU-only*. For instance, for YCSB A workload under skewed key distribution (Figure 5(c)), *CPU-only* and *DPU-only* achieve latencies of 80μs and 69.3μs, with throughput being 28.6 Mops and 34.8 Mops, respectively. These numbers indicate that when DPU's SoC subsystem (CPU + memory) is more powerful than the edge host, as with BF-3, *DPU-only* outperforms *CPU-only* by achieving both higher throughput and lower latency. On the other hand, when the DPU's SoC cores are weaker than the edge host, as with the BF-2 DPU, coarse-grained offloading (*DPU-only*) at comparable latency fails to match the throughput of CPU-only KVS. Based on these observations, we are motivated to profile the performance of a monolithic KVS to identify decomposable components (i.e., the C1 challenge in Figure 1), which are suitable for fine-grained offloading to DPUs.

3.3 Profiling and Decomposition of KVS

As depicted in Figure 4 and discussed in § 3.2, we logically divide a CPU-based monolithic KVS into two major components—a processing engine and a communication engine. The KVS processing engine performs two major functions—`GET` and `SET`. These two functions either retrieve or store a key-value pair in memory. The communication engine performs three basic functions. The `receive` function retrieves network packets from the NIC's receive queue. The `parse request` function is responsible for extracting the key-value request from the raw network packet. The `response` function allocates and prepares the network buffer, fills it with the KV response and other packet metadata, and finally sends out the packet to the client. We identify which of these finer-grained KVS components—communication or processing engine—are CPU-intensive by breaking down their host CPU usage and the time spent by a KV request batch in each KVS component.

Table 1: CPU usage of various components in CPU-only KVS (MICA) server with BF-2/3 for different YCSB workloads. Communication is CPU-intensive consuming 63%–70% host CPU cycles. BF2 and BF3 are used as regular NICs for the CPU-only case.

		CPU usage (%)					
Component (engine)	**Host Function**	YCSB A		YCSB B		YCSB C	
		BF2	BF3	BF2	BF3	BF2	BF3
Packet Processing (comm.)	receive	18.2	25.4	16.4	23.7	14.4	17.2
	parse	17.1	20.7	21.9	20.6	23.9	23.4
	response	27.5	23.2	28.9	25.3	29.8	28.1
Key-value Processing	Get/Set	19.5	13.4	15.9	10.2	13.1	9.5
	others	17.7	17.2	16.6	20.1	18.5	21.7

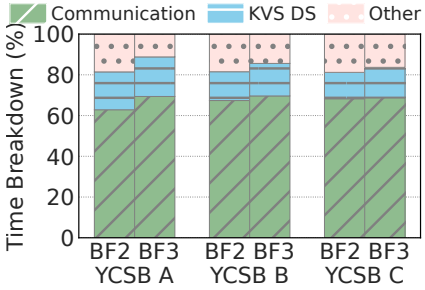
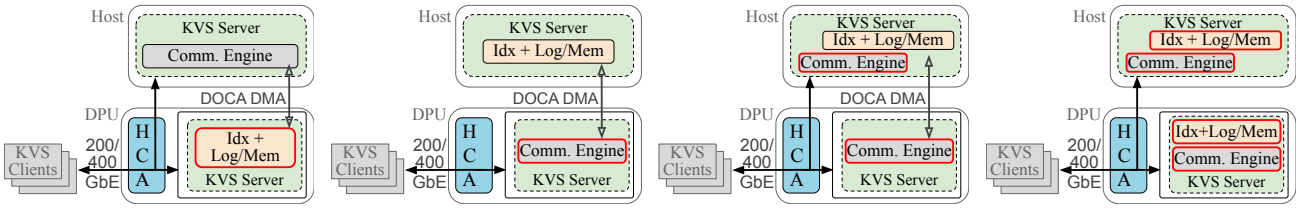


Figure 6: KVS time breakdown for CPU-only KVS with BF-2/3 on a batch of 64 requests. Processing engine accounts for 10%–19% of KV batch processing time. 'KVS DS' represents the time spent manipulating KV 'Data Structures' in the processing engine.

Table 1 and Figure 6 show the host CPU usage and KV processing time across different KVS components in a CPU-only configuration (i.e., on the edge host), using YCSB workloads with uniformly distributed keys. CPU usage is measured with the Linux `perf` tool [44]. The 'YCSB A BF2' column of Table 1 shows that the KV processing engine consumes up to 19.5% of total CPU cycles on BF-2. On BF-3, it accounts for up to 13.4%, as indicated in the 'YCSB A BF3' column. In Figure 6, the blue segments show that a KV request batch spends 10%–19% of its total time in the processing engine on BF-2 and BF-3.

For the communication engine, Table 1 shows that both direct (`receive, response`) and indirect (`parse request`) packet processing consume up to 70% of host CPU cycles—up to 3.5× more than the processing engine—on both BF-2 and BF-3. Of the total CPU cycles consumed by the communication engine (Table 1), up to 37%—specifically, 25.4 out of 69.3 cycles for the YCSB A workload on CPU-only KVS using BF-3—are spent polling the NIC's receive queue for incoming packets. Additionally, Figure 6 shows that a KV request batch spends, on average, 4.3× more time on communication than on processing for various workloads on BF-2/3.

Thus, we can safely conclude that the communication engine is the most CPU-intensive component of a monolithic KVS on the edge



(a) Processing engine offload (b) Communication engine offload (c) Dual communication engine (d) Sharding-based design

Figure 7: Various fine-grained offloading architectures for KVS server by modularizing KVS into processing and communication engine. a) Processing engine offloaded to DPU. b) Communication engine offloaded to DPU. c) Processing engine on the host with the communication engine on both the host and DPU. d) Entire KVS runs on both host and DPU (i.e., sharing-based design).

host. Based on this insight, we are motivated to further investigate whether—and how—fine-grained KVS offloading to DPUs can enhance performance, which we explore next.

4 Fine-Grained KVS Offloading to DPU

In this section, we explore effective strategies for fine-grained KVS offloading to DPUs to improve performance.

4.1 Fine-Grained Offloading Architectures

Figure 7 shows various KVS server offloading strategies, achieved by decomposing the KVS into separate processing and communication engines. This modular design enables selective offloading of the processing engine (Figure 7(a)), the communication engine (Figure 7(b) and Figure 7(c)), or both (Figure 7(d)) to the DPU.

Based on our analysis in § 3, we focus on offloading the KVS communication engine to the DPU (C1 in Figure 1) for three key reasons. First, § 3 shows that the communication engine in CPU-only KVS consumes over two-thirds of host CPU cycles, making it the most critical component to offload in order to free up CPU resources on the edge host. Second, the item set size of in-memory KVS must be considered when deciding what to offload. Prior works [3, 36, 64, 69] have shown that in-memory KVS item sets are typically on the order of the host's DRAM capacity. This makes it impractical to store all key-value items on DPUs, which have significantly less memory than host CPUs. Third, the memory requirement of the communication engine (e.g., 1–4 GB for Data Plane Development Kit (DPDK) [21] as used in MICA) is well within the 16 GB DRAM capacity of NVIDIA's BF-2/3 DPUs[59, 60].

Thus, offloading the KVS communication engine would free up more host CPU cycles, which can be utilized by the KVS processing engine to process more KV requests.

A hybrid KVS design is also possible where KVS operations (`GET`/`SET`) are offloaded to DPUs as discussed in KV-Direct [36]. A disadvantage of such a design is that it requires complex memory management schemes to keep KV pairs synchronized across DPU memory and host memory, as the stored keys are partitioned across them. Such a hybrid design approach is complementary to our offloading design, and we wish to study combining hybrid designs with offloading-based approaches in the future.

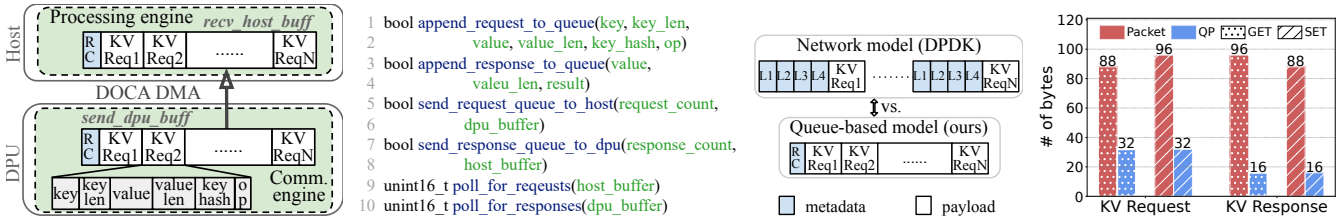
Moreover, our exploration indicates that offloading the communication engine to the DPU is non-trivial, primarily due to several challenges, including C2 (efficient host-DPU data path), C3 (minimized data movement overhead), C4 (reduced host resource usage), and C5 (optimized performance), which must be carefully addressed. These challenges are detailed in our exploration journey, illustrated in Figure 1. To solve C2, we discuss how to design a low-latency data path between the host and the DPU, enabled by our key-value-based Queue-Pair model, as described in § 4.2. To address C3, we propose an overlapped KV request/response stage along with additional latency optimizations, as discussed in § 4.3. For C4, we adjust the processing-to-communication engine core mapping between the host and the DPU, as discussed in § 4.4. Finally, to address the goal of C5, we explore an additional design featuring dual communication engines in § 4.5, and evaluate the performance trade-offs among various offloading architectures (Figure 7) in § 6. We refer to our fine-grained offloading approach as "DPU-KV", which accelerates CPU-based KVS (*CPU-only*) by offloading specific KVS tasks or components to the DPU.

4.2 Low-Latency Host-DPU Data Path

To study the efficacy of DPU-offloaded KVS, it is essential to establish a fast and efficient data path that helps KVS components on the host and DPU communicate and process KV requests together. We first study how to facilitate an efficient key-value data exchange between offloaded (communication engine) and non-offloaded (processing engine) KVS components and propose a set of APIs to achieve this. Next, we analyze diverse low-level DMA primitives offered by DPUs to establish a low-latency host-DPU data path.

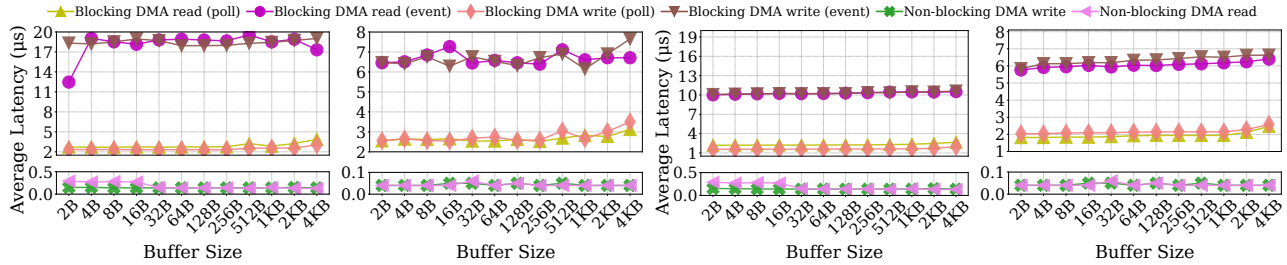
4.2.1 Host-DPU Key-Value-based Queue-Pair Model. The Queue-Pair (QP) based programming model is popular in many networking and storage systems [8, 47, 78] for its high performance. We adopt the same QP model to maintain a dedicated path between the host and DPU. We further extend the model to support key-value data exchange, which we refer to as a KV-based QP model.

Our KV-based QP model contains send and receive queues on the host and DPU for key-value data exchange. Figure 8(a) depicts the structure of request QP. A response QP follows a similar structure. A complete list of APIs that facilitate KV data exchange is listed in Figure 8(b). The DPU, after parsing the KV request from the network packet, appends the relevant fields (e.g., key, key length, value, value length, key hash, etc.) to the request QP as shown in Figure 8(a). Other KV fields (e.g., request timestamp and packet headers) are stored on the DPU to help prepare the response



(a) Request queue-pair (b) List of APIs (c) Metadata of prog. models (d) Bytes transferred across PCIe

Figure 8: Overview of KV-based QP model and its benefits. a) KV data exchange between the host and the DPU. RC (one byte) denotes the total request/response count. We set the max key length to 8 bytes, with KV requests and responses sized at 32 and 16 bytes, respectively, following MICA's defaults; b) List of APIs, c) Metadata comparison of DPDK's network model and our proposed KV-based QP model, and d) Bytes transferred across PCIe for CPU-only KVS vs. KV-based QP model. KV QP reduces data/metadata transfer for KV requests/responses.



(a) DPU-initiated (BF-2) (b) Host-initiated (BF-2) (c) DPU-initiated (BF-3) (d) Host-initiated (BF-3)

Figure 9: DOCA DMA latency of different primitives on BF-2/3 as payload increases. Non-blocking DMAs achieve lowest latency.

packet. This approach minimizes the DMA buffer size and reduces unneeded PCIe traffic between the host and the DPU. To avoid multiple data copies, the host and DPU perform request and response processing directly on the request/response queue pairs. Each host core and DPU core maintains its own request and response queue pairs to avoid inter-core communication overheads [26, 43, 75]. The KV-based QP model also requires minimal metadata, in contrast to the network model used by CPU-only KVS, which includes additional metadata such as packet headers (Layer 1–Layer 4) for each KV request/response, as shown in Figure 8(c).

We compare the data transfer efficiency of the KV-based QP model with the network packet model used by CPU-only KVS. We measure the number of bytes transferred across the PCIe per 8 B key-value request/response for CPU-only KVS vs. the KV-based QP model used by DPU-offloaded KVS. From Figure 8(d), we observe that our proposed KV-based QP model can save up to 63.67%/83.3% and 66.67%/81.81% bytes per network packet for requests/responses in `GET` & `SET`, respectively.

Thus, a key-value-based queue-pair model enables efficient data transfer over PCIe when offloading the KVS component to DPUs.

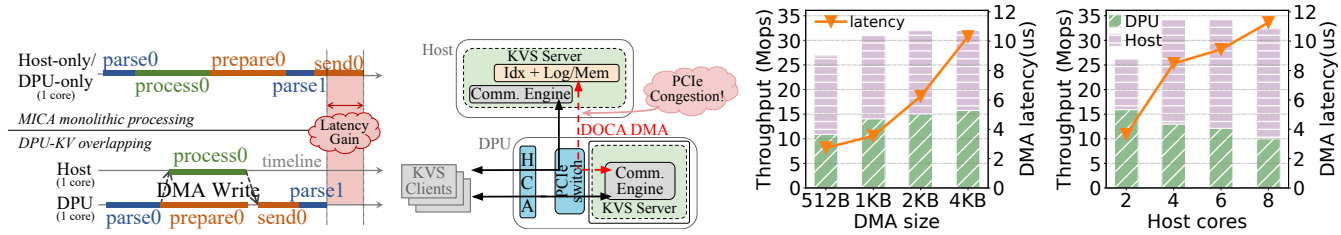
4.2.2 Low-level DPU DMA Primitives. To enable fast host-DPU data movement, our APIs—`send_request_queue_to_host` and `send_response_queue_to_dpu`—internally use NVIDIA's DOCA DMA APIs [58]. The DMA APIs allow the SoC on the DPU and the host to utilize the DPU's DMA engine to help move memory buffers across the PCIe interface. Given the different ways one can utilize these DMA APIs—querying for DMA completions (blocking, either through polling or event-driven approaches vs. non-blocking) and initiator of DMA request (Host vs. DPU)—understanding their performance is imperative.

We analyze the performance of different low-level DMA primitives supported by DPUs and choose the one with the lowest latency for our APIs. Figure 9 measures DMA latency for different DMA operations for varying buffer sizes on BF-2/3. The experiment uses a single core on both the host and the DPU to measure DMA latency and report the average latency of 5000 DMA operations. Figure 9 shows that the non-blocking DMA reads/writes have the lowest latency compared to blocking DMA operations, irrespective of whether the host or the DPU initiates the operation. This is because we do not need to check the completion of each KV request, allowing for better overlap. However, end-to-end request completion still guarantees that all intermediate operations, including non-blocking DMA, have completed successfully.

Thus, our host-DPU data path uses non-blocking DMA writes and key-value-based QP model for low latency and efficient key-value data exchange. The proposed low-latency host–DPU data path is applied across all subsequent DPU-KV offloading designs.

4.3 Hiding Data Movement Penalty

KVS request processing for a request batch can be divided into three stages—parse, process, and response (prepare and send). The top portion of Figure 10(a) shows the existing request processing pipeline when the entire KV processing occurs on the host or the DPU. The bottom portion of Figure 10(a) illustrates the proposed "*DPU-KV-lat*" pipeline, highlighting where each KV processing stage executes when the KVS communication engine is offloaded to the



(a) KV request/response processing pipeline in MICA and DPU-KV

(b) Host-DPU communication bottleneck in *DPU-KV-dual* due to PCIe congestion

(c) Increasing PCIe congestion w.r.t. DMA buffer size

(d) Increasing PCIe congestion w.r.t. number of clients/host cores

Figure 10: Optimizations in "*DPU-KV*" designs. a) Proposed designs for improving latency via overlapped key-value processing. b) The red line indicates increased host-DPU DMA latency due to host PCIe congestion. *DPU-KV-dual* total throughput and DMA latency by varying c) DMA buffer size, and d) concurrent clients for YCSB C workload. Host PCIe congestion limits *DPU-KV-dual's* throughput.

DPU (Figure 7(b)). The packet processing stages, namely, the parsing of requests and preparing/sending responses, now run on the DPU while the KV processing remains on the host. The proposed DPU-KV-lat design incorporates various optimizations to alleviate data movement overheads, which we discuss in detail below.

Reducing DMA operations per batch. CPU-/DPU-only KVS processes each request in a batch to completion before processing the next request in the same batch. If we apply this approach while offloading, the communication engine on the DPU would need to issue a DMA write for each request in the batch, increasing PCIe traffic. Also, Figure 9 shows that the latency of a DMA write operation for a 2 B payload vs a 2 KB payload is similar. Thus, we parse all the requests in the request batch instead of parsing them individually. *DPU-KV-lat* issues a single DMA write per request batch to send necessary key-value data to the host for processing. Similarly, the host sends the entire batch of processed KV requests, i.e., KV responses, back to the DPU with a single DMA write operation.

Overlapped KV request/response processing. We also overlap KV processing stages between the host and DPU to reduce overall response time. On DPU, a simple approach is to start the response stage upon receiving the responses from the host for the previous request batch. This keeps DPU idle while waiting for the host to process and send KV responses. To further improve performance, we utilize idle time to prepare response packets by splitting the prepare response stage from the send response stage. In the prepare response stage, the DPU performs the following tasks—a) allocates network buffers for new requests and frees the ones that have been processed, and b) fills the response packet with other data related to KV items like key hash and the request timestamp, and packet-related information like source/destination addresses, packet length, etc. By cleverly overlapping the prepare response stage in the DPU with the process stage in the host, we further reduce the overall request processing time as shown in Figure 10(a).

Response processing optimization. In *CPU-/DPU-only* response processing pipelines, responses from the previous request batch are sent only after the next batch has been parsed. For example, in the top part of Figure 10(a) (labeled "MICA monolithic processing"), `send0` occurs after `parse1` in CPU-/DPU-only pipelines. When offloading the communication engine to the DPU, following this approach causes KV responses from the previous batch to be delayed until the next batch is processed, increasing overall latency. To mitigate this, we modify the pipeline to initiate the send response stage as soon as the responses are ready on the DPU, before parsing the next batch. As shown in the lower part of Figure 10(a) (labeled "DPU-KV overlapping"), `send0` occurs before `parse1` in the DPU-KV pipeline.

4.4 Reducing Host Resources

In the DPU-KV-lat design presented in § 4.3, each host core running the KVS processing engine is mapped one-to-one to a DPU core running the communication engine. This dedicated host-to-DPU core pairing ensures parallel access to individual KV partitions and allows each pair to maintain separate DMA buffers, avoiding inter-core communication and synchronization overhead. While the DPU-KV-lat design achieves low latency, its one-to-one core mapping leads to high host core utilization, comparable to that of the CPU-only KVS.

To reduce host resource usage in edge environments, the mapping between host processing cores and DPU communication cores can be transitioned from a one-to-one to a one-to-many model. We propose a new design, "*DPU-KV-sav*", which leverages the insight that offloading the CPU-intensive communication engine frees host CPU cycles. This allows the processing engine on the host to handle a larger number of KV requests, thereby reducing the number of host CPU cores needed to service incoming requests.

In the DPU-KV-sav design, a single host core running the processing engine maintains exclusive access to multiple DPU cores and processes KV requests in a round-robin manner. While this design currently uses simple scheduling, more advanced scheduling strategies could be explored in the future. The host core maintains dedicated DMA buffers for each DPU core, similar to *DPU-KV-lat*, allowing *DPU-KV-sav* to significantly lower host CPU utilization.

4.5 Performance with Resource Savings

We observe that the throughput of *DPU-KV-lat* and *DPU-KV-sav* with BF-2, though higher than the DPU-only KVS, is still lower than CPU-only KVS (§ 6.2). To improve the throughput of DPU-offloaded KVS while utilizing minimum host resources and maintaining low latency, we propose a dual communication engine-based design ("*DPU-KV-dual*") as shown in Figure 7(c). *DPU-KV-dual* leverages

both the DPU cores and spare host cores (enabled by reduced host resource usage from § 4.4) to receive and process requests for the same KVS memory backend. One host core continues to process requests from DPUs (main host core), similar to the DPU-KV-sav design, while the spare cores expose themselves as additional endpoints to the client. These spare host cores run the communication engine and perform KV processing. The additive throughput of main and spare CPU cores allows *DPU-KV-dual* to attain modestly better throughput than CPU-only KVS at lower host core utilization.

To determine the necessary number of spare host cores, we evaluate the throughput of *DPU-KV-dual* while varying the number of spare host cores as shown in Figure 10(d). We find that a total of three host cores (one main and two spare host cores) are sufficient for *DPU-KV-dual* to reach peak throughput. Using more spare host cores only increases host CPU utilization without increasing throughput due to host PCIe congestion, as explained below.

Host PCIe congestion analysis. We find host PCIe congestion to be the main bottleneck for *DPU-KV-dual* throughput stagnation as more host cores are added. *DPU-KV-dual* runs two communication engines—one on the DPU and the other on the host.

The communication engine on the DPU sends the extracted KV requests to the host. Meanwhile, the communication engine on the host receives raw packets containing KV requests from the clients, increasing PCIe traffic to/from the host and causing congestion, as shown in Figure 10(b).

We observe that the PCIe congestion on the host depends upon two factors—DMA buffer size and concurrent connections with the client. We quantify this congestion by measuring per-batch host-DPU DMA latency by fixing one factor and varying the other. We find that increased DMA latency between the host and DPU corresponds to higher PCIe congestion. This shows that PCIe often becomes a bottleneck under heavy data loads, underscoring the need to optimize data paths in offloaded KVS designs. Figure 10(c) shows how total throughput and DMA latency in *DPU-KV-dual* vary with increasing DMA size while fixing the number of client-host connections to two. The total throughput stagnates when DMA size is $> 1\,KB$. This is because the slight increase in throughput provided by DPU (4.7%) is offset by the decrease in throughput by the host (4.1%) when DMA buffer size increases from 2 KB to 4 KB. Therefore, increasing the DMA size while maintaining fixed client-host connections limits the overall system throughput. The larger DMA size results in more data traversing the host PCIe, negatively impacting the host throughput.

In Figure 10(d), the DMA buffer size is fixed at 2 KB. We vary the number of concurrent client connections to the host and measure throughput and DMA latency in *DPU-KV-dual*. We observe that the total throughput is almost constant after using a total of 4 host cores. As the number of connections increases, the throughput by DPU reduces by 36.6% as DMA latency increases by 3.1x, limiting the overall throughput of the system. The reason for this peculiar behavior is the increase in DMA latency as host PCIe congestion increases with the increase in the number of client connections.

Overall, the above findings show that the data movement cost between the DPU and host is not fixed and is proportional to host PCIe congestion.

To further improve throughput, it is important to mitigate PCIe congestion between the host and the DPU. Figure 7(d) presents a sharding-based design, referred to as "*DPU-KV-shrd*," in which the key-value store is partitioned and executed independently on both the host and the DPU. By avoiding host-DPU communication during request processing, this design reduces PCIe traffic and allows each side to operate on disjoint data partitions, improving overall system parallelism and efficiency. In § 6.2, we demonstrate that the sharding design with BF-2 achieves higher throughput than CPU-only KVS, though at the cost of high host resource consumption.

5 Implementation Tips

In this section, we share our experience implementing DPU-offloaded KVS on BF-2 and BF-3 DPUs using the latest version of state-of-the-art in-memory KVS, MICA2 [42].

Connection Management. MICA2 uses etcd [24] on the server side to expose network endpoints. During initialization, the KVS client reads the endpoint information from the etcd daemon and starts sending key-value requests to those endpoints. DPU registers its endpoints with etcd running on the server. *No code changes are required for the MICA2 client.*

Porting. MICA2 is built with DPDK v16.11. We port MICA2 to run with DPDK v21.08. MICA2 uses Flow Director (FDir) [18, 22] APIs in DPDK for request direction (`rte_eth_dev_filter_ctrl()`), which is deprecated since v20.11 [19]. We replace deprecated FDIR APIs with `rte_flow` APIs (`rte_flow_validate()` and `rte_flow_create()`) [20] for flow direction.

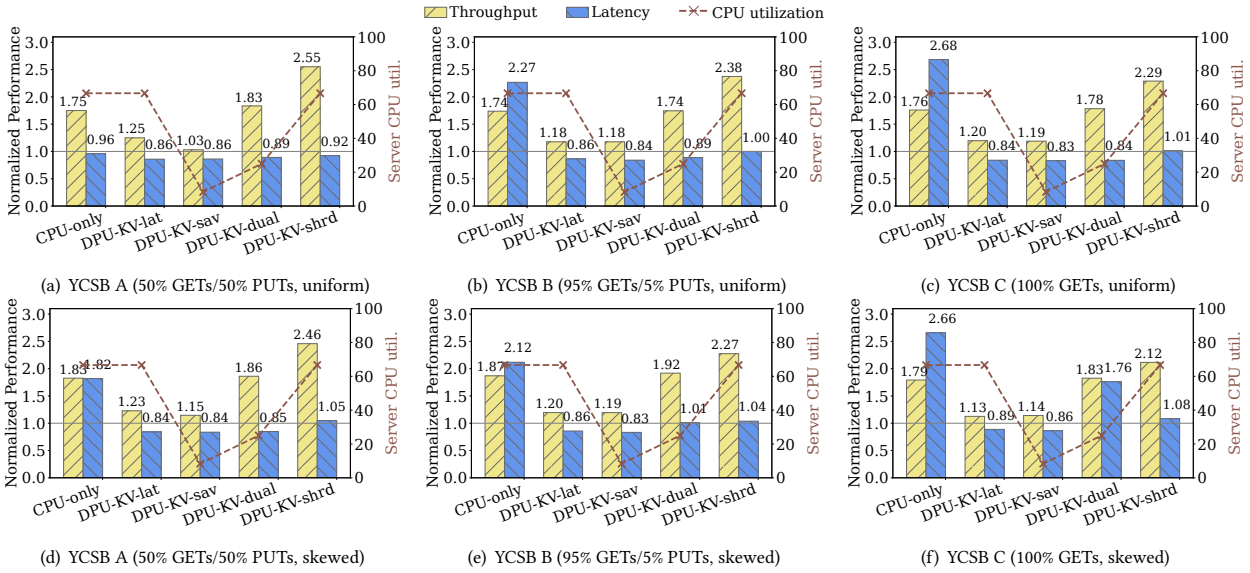
MICA2 uses x86 instructions for certain operations. This hinders running MICA2 directly on other architectures like ARM on DPUs. To run MICA2 on SoC-based DPUs, we replace x86-specific instructions with ARM architecture ones. x86 `PAUSE` and `RDTSC` instructions are replaced with ARM's `YIELD` and `CNTVCT_EL0` instructions, respectively.

DOCA DMA. Our key-value-based QP model utilizes DPU's DMA engine for host-DPU KV data exchange. The DMA APIs, provided through DOCA SDK [62], varied slightly between DOCA SDK versions used by our BF-2 and BF-3. The major API changes were in how the host/DPU communicates with the hardware DMA engine (`doca_workq` for BF-2 vs. `doca_pe` for BF-3), submits DMA jobs (`doca_workq_submit()` for BF-2 vs. `doca_task_submit()` for BF-3), and queries their completion status (`doca_workq_progress_retrieve()` for BF-2 vs. `doca_pe_progress()` for BF-3). Additionally, `doca_pe` introduced a task completion callback, requiring the reset of the DMA buffer length for reusing the DMA buffer in subsequent DMAs.

Adding timestamp to KVS requests. MICA2 does not report key-value request latency. We enable latency tracking of requests by tagging each KV request with its timestamp when a request is generated by the KVS client. Similar to [43], we compare the current timestamp with one echoed back from the server in the KV response to measure latency on the KVS client.

6 Evaluation

Our evaluation answers the following questions regarding DPU-offloaded in-memory KVS at the edge— ❶ How much performance gain in terms of latency and throughput does DPU-offloaded KVS



(a) YCSB A (50% GETs/50% PUTs, uniform) (b) YCSB B (95% GETs/5% PUTs, uniform) (c) YCSB C (100% GETs, uniform)

(d) YCSB A (50% GETs/50% PUTs, skewed) (e) YCSB B (95% GETs/5% PUTs, skewed) (f) YCSB C (100% GETs, skewed)

Figure 11: Peak performance of KVS designs with BF-2 DPU showing latency, host utilization, and throughput for YCSB A, B, and C workloads. The top ((a)-(c)) and bottom ((d)-(f)) rows represent uniform and skewed key distribution, respectively. Here, DPU indicates BF-2. Performance of *CPU-only*, *DPU-KV-lat*, *DPU-KV-sav*, *DPU-KV-dual*, and *DPU-KV-shrd* normalized to their respective DPU-only case. *DPU-KV-shrd* attains highest throughput. *DPU-KV-dual* achieves a balanced trade-off between performance and resource efficiency.

get compared to CPU-based KVS when offloading to BF-2 (§ 6.2) and BF-3 (§ 6.3)? ❷ Do DPU-offloaded KVS designs scale with DPU cores (§ 6.4)?

6.1 Experimental Setup

Our platform details and KVS configurations are described in § 3.1. We compare the performance of DPU-offloaded KVS with two baselines—*CPU-only* and *DPU-only*. *CPU-only* represents the performance of MICA KVS when the entire KVS runs on the host without any offloading to DPU. In *CPU-only*, the DPU (BF-2 or BF-3) on the server node acts like a NIC serving KV requests from clients. The *DPU-only* case represents a coarse-grained offloading approach, where the entire KVS (MICA) is offloaded to the DPU. The details of porting MICA to run on the ARM architecture of BF-2/BF-3 and augmenting MICA's KV request generator to measure latency are described in § 5.

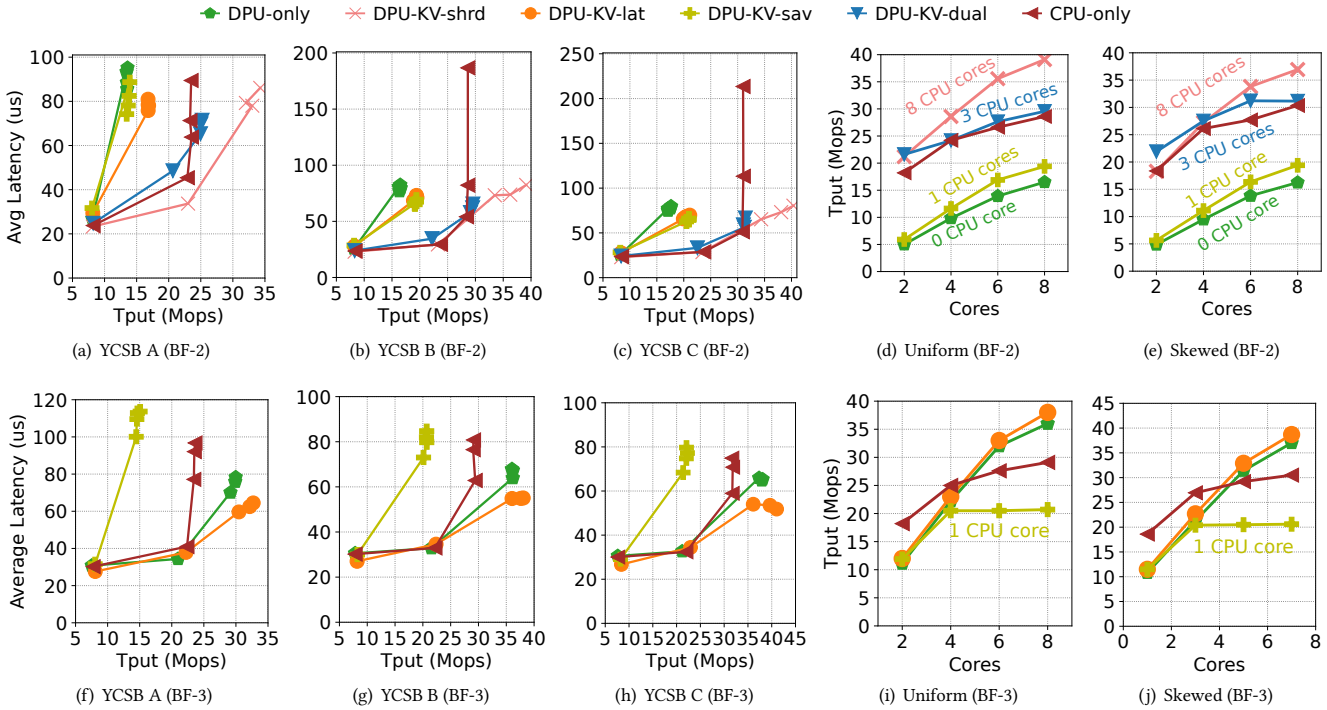
As defined in § 4, among the DPU-offloaded KVS, *DPU-KV-lat* optimizes for latency by mitigating the data movement penalty of offloading, and *DPU-KV-sav* design represents saving host resources. *DPU-KV-dual* represents throughput optimization over *DPU-KV-sav*, and *DPU-KV-shrd* is a high throughput design. BF-2 supports *DPU-KV-lat*, *DPU-KV-sav*, *DPU-KV-shrd*, and *DPU-KV-dual*, whereas BF-3 supports *DPU-KV-lat* and *DPU-KV-sav*. This is because, in the DPU mode of BF-3, the host and the SoC cores cannot operate network interfaces simultaneously (§ 2.1).

To keep the comparison fair, we allocate a total of 12 GiB of hugepages for the KVS server across all offloaded and non-offloaded designs. The KVS server uses 2 GiB of hugepages for DPDK [21] on either host or DPU, based on whether the communication engine is offloaded or not. *DPU-KV-sav* utilizes a single host core. *DPU-KV-dual* employs three host cores, as this configuration offers an optimal balance between throughput and host core utilization, as detailed in § 4.5. Unless otherwise stated, *CPU-only*, *DPU-KV-shrd*, and *DPU-KV-lat* use 8 host cores, while all other designs, except *CPU-only*, leverage all 8 SoC cores. In all cases, the packet burst size in DPDK (maximum packets transmitted/received) is set to 64 for BF-2 and 256 for BF-3. The DPU-offloaded designs pre-allocate and pre-register DMA buffers per core as follows: 2049 bytes ($64*32+1$) and 1025 bytes ($64*16+1$) when using BF-2, and 8193 bytes ($256*32+1$) and 4097 bytes ($256*16+1$) when using BF-3. Here, 32 B and 16 B represent the size of KV request/response in the request and response queue pairs, respectively, while 1 B represents the request/response count.

6.2 Offloaded KVS Performance with BF-2 DPU

Figure 11 depicts the peak throughput, latency, and host CPU utilization of baseline and DPU-offloaded designs with BF-2 for different YCSB workloads following uniform and skewed key distributions. In Figures 12(a), 12(b), and 12(c), we also plot end-to-end latency as a function of throughput by varying the client request generation rate for uniform key distribution for write-heavy, read-heavy, and read-only workloads, respectively. In this subsection, the term DPU specifically refers to the BF-2 DPU.

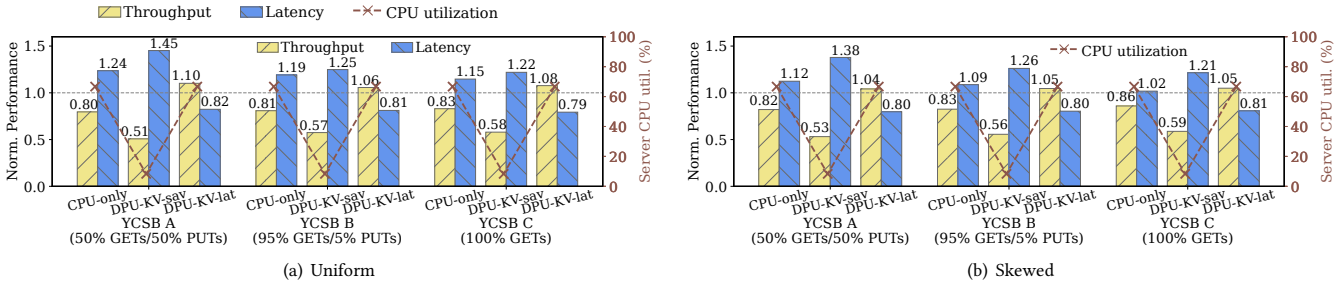
DPU-KV-lat reduces latency by 11%–16% and 10.4%–68.7% over *DPU-only* and *CPU-only* designs, respectively, for different workloads and key distributions, as seen in Figure 11. Communication engine offloading and implementing various latency optimizations help *DPU-KV-lat* reduce the latency over *DPU-only* and *CPU-only* KVS. The reduction in latency helps increase *DPU-KV-lat* throughput by 1.2× on average over DPU-only KVS across all workloads and key distributions, as shown in Figure 11 and Figure 12(a) – 12(c).



(a) YCSB A (BF-2) (b) YCSB B (BF-2) (c) YCSB C (BF-2) (d) Uniform (BF-2) (e) Skewed (BF-2)

(f) YCSB A (BF-3) (g) YCSB B (BF-3) (h) YCSB C (BF-3) (i) Uniform (BF-3) (j) Skewed (BF-3)

Figure 12: End-to-end latency vs. throughput of KVS designs for uniform distribution for YCSB workloads. Sub-figs(a,b,c) and Sub-figs(f,g,h) indicate KVS with BF-2 and BF-3. Throughput of KVS designs when varying numbers of cores for uniform/skewed distribution for YCSB B workload. Similar trend observed for other YCSB workloads. Sub-figs(d,e) and Sub-figs(i,j) indicate KVS with BF-2 and BF-3. For BF-2, we omit *DPU-KV-lat* as its scaling performance is similar to *DPU-KV-sav*.



(a) Uniform (b) Skewed

Figure 13: Peak performance of KVS designs with BF-3 DPU showing latency, host utilization, and throughput for YCSB workloads. (a) Uniform and (b) Skewed key distribution. In this figure, DPU indicates BF-3. Performance of *CPU-only*, *DPU-KV-sav*, and *DPU-KV-lat* normalized to their respective DPU-only case. *DPU-KV-lat* achieves the best performance on BF3.

The DPU-KV-lat design follows a one-to-one host-to-DPU core mapping of KVS processing-to-communication engine consuming 66.67% of host cores as seen in Figure 11. The one-to-many processing-to-communication engine mapping in *DPU-KV-sav* utilizes only a single host core for the processing engine, bringing down the host CPU utilization to 8.33% while maintaining low latency. For instance, *DPU-KV-sav* reduces latency by 14%–17% and 11.1%–68.9% over DPU-only and CPU-only designs, respectively, for different workloads and key distributions, as seen in Figure 11.

The similar performance of *DPU-KV-sav* compared to *DPU-KV-lat*, despite 87.5% lower host core utilization, indicates that the freed host cycles via offloading can be used to run the non-offloaded KVS component (processing engine) on fewer host cores. Additionally, the latency optimizations of *DPU-KV-lat* remain effective in *DPU-KV-sav* when the DPU-to-host core mapping is modified. Though *DPU-KV-sav* saves maximum host resources among DPU-offloaded KVS, its throughput is 32%–41% lower than *CPU-only*, as seen in Figure 11. This is because the wimpy SoC cores of BF-2 running the offloaded communication engine are unable to fully saturate the KVS processing engine on the host.

As shown in Figure 12(a) – 12(c), *DPU-KV-dual* achieves peak throughput of 25.20 Mops, 29.57 Mops, and 31.43 Mops for uniform

key distribution for YCSB A, B, and C workloads, respectively. *DPU-KV-dual* achieves 1.74×–1.92× and 1.47×–1.6× higher throughput over *DPU-only* and *DPU-KV-sav*, respectively. The throughput increase in *DPU-KV-dual* comes from spare host cores (freed up by DPU-KV-sav design) handling up to 16 M additional KV requests (as discussed in § 4.5). *DPU-KV-dual's* throughput is comparable to *CPU-only*, with its throughput 1.01×–1.07× that of the *CPU-only*, as seen in Figure 12(a) – 12(c). However, *DPU-KV-dual* achieves this throughput at 33.8%–68.6% lower latency and 62.5% lower host core utilization than *CPU-only* KVS, as shown in Figure 11. The reduced latency of *DPU-KV-dual* is due to the main host core processing KV requests from the DPU, similar to the DPU-KV-sav design, thereby benefiting from its latency optimizations.

Figure 11 also demonstrates that *DPU-KV-shrd* consumes 2.67× higher host cores than *DPU-KV-dual*, achieving 1.21×–1.45× higher throughput than *CPU-only*, indicating sharding-based design can help enhance throughput over CPU-only KVS.

Therefore, for DPU-offloaded KVS with BF-2, the DPU-KV-shrd design is recommended when high throughput is required. When host resources are limited for KVS in the edge, DPU-KV-dual should be used. DPU-KV-dual provides a balanced performance with lower latency and comparable throughput while consuming fewer host resources than CPU-only KVS.

6.3 Offloaded KVS Performance with BF-3 DPU

In this section, we compare the performance of baseline and DPU-offloaded KVS with BF-3 as shown in Figure 13. Figures 12(f), 12(g), and 12(h) depict latency vs. throughput for uniform key distribution for write-heavy, read-heavy, and read-only workloads, respectively. In this section, the term DPU refers to the BF-3 DPU.

CPU-only exhibits lower performance than *DPU-only*. For instance, *CPU-only* has up to 20% lower throughput and 24% higher latency than *DPU-only* across all workloads, as shown in Figure 13. This is because the SoC subsystem (CPU and memory) of BF-3 is brawnier than the edge host, enabling it to process more KV requests. With BF-2, *DPU-KV-sav* outperforms *DPU-only* (as discussed in § 6.2), but this is not the case with BF-3. As seen in Figure 13, with BF-3, *DPU-KV-sav* shows up to 49% lower throughput and 45% higher latency than *DPU-only* across various workloads. This occurs because, when the communication engine runs on BF-3 instead of BF-2, the processing engine on the host core becomes the bottleneck.

DPU-KV-lat attains the best performance compared to CPU-/DPU-only KVS designs. For example, for the YCSB C workload, the peak throughput of *DPU-KV-lat* is 41.03 Mops (vs. 38 Mops of *DPU-only*) and latency is 21% lower than *DPU-only* (51.8 μs vs. 65.3 μs of *DPU-only*), as shown in Figure 12(h). Across all workloads, *DPU-KV-lat* shows up to 33.3% and 21% lower latency, along with up to 35.7% and 10% higher throughput, compared to *CPU-only* and *DPU-only*, respectively, as seen in Figure 13. Offloading the communication engine and optimizing latency (fewer DMA operations, overlapped KV processing, and response optimization) helps *DPU-KV-lat* to achieve better performance than *CPU-/DPU-only*.

Thus, for DPU-offloaded KVS with BF-3, the DPU-KV-lat design should be utilized when edge applications utilizing KVS require low latency and high throughput.

6.4 Scalability

Here we compare the scalability of DPU-offloaded KVS designs with CPU-/DPU-only KVS. We measure the throughput while varying the number of host CPU cores for *CPU-only* and DPU ARM cores for DPU-based designs.

Figure 12(d) and Figure 12(e) illustrate the scalability of baseline and DPU-offloaded KVS with BF-2 for YCSB B for both uniform and skewed workloads, respectively. For uniform workloads, *DPU-KV-sav*, *DPU-KV-dual*, and *DPU-KV-shrd* scale in terms of throughput as the number of DPU cores increases. This trend continues for skewed workloads as well, except for the DPU-KV-dual design. The *DPU-KV-dual* reaches peak throughput faster than *CPU-only* (at six ARM cores), which is its upper limit due to host PCIe congestion.

Similarly, Figure 12(i) – 12(j) show the scalability of baseline and DPU-offloaded KVS with BF-3 for uniform/skewed workloads. *DPU-KV-sav* rapidly reaches saturation around 20.5 Mops across both workloads when the number of ARM cores is four or more. This indicates that the communication engine running on four BF-3 ARM cores is sufficient to saturate the processing engine operating on a single host CPU core, unlike *DPU-KV-sav* with BF-2, which exhibits linear scalability. In contrast, *DPU-KV-lat* employs a one-to-one core mapping between the processing and communication engine, yielding throughput that scales linearly with the number of ARM cores (and host cores) across both workloads.

The linear scalability of *DPU-KV-sav* and *DPU-KV-dual* with BF-2, and *DPU-KV-lat* with BF-3, shows that our host-DPU data path scales well while minimizing PCIe transfers. Overall, DPU-offloaded KVS scales effectively for both uniform and skewed workloads, similar to CPU-/DPU-only designs.

7 Conclusion & Future Work

This paper explores different DPU offloading strategies for in-memory KVS, including coarse-grained and fine-grained designs. We find that DPUs can enhance KVS performance at the edge through various fine-grained offloading approaches. To achieve this, we have systematically identified the CPU-intensive KVS component, the KVS communication engine, as a prime candidate for offloading. Also, we introduce several optimization techniques to minimize data movement overheads and boost performance. We test our DPU-KV designs with NVIDIA BF-2 and BF-3 DPUs. Our experiment results with BF-2 indicate that DPU-offloaded KVS can reduce latency by up to 68% and host core usage by 62.5% over MICA (*CPU-only*) KVS while achieving modestly higher throughput. Our results with BF-3 DPU show latency reduction by 33.3% and 21%, along with throughput improvement by up to 35.7% and 10% compared to CPU-only and DPU-only KVS. In the future, we aim to explore offloading additional KVS tasks, such as replication and/or consistency, to DPUs at the edge.

Acknowledgments

We would like to thank all the anonymous reviewers for their valuable feedback. We gratefully acknowledge NVIDIA for their generous donation of BlueField 2 and BlueField 3 DPU hardware resources. This work was partly supported by NSF research grants OAC #2321123 and #2340982, an Amazon Faculty Research Award, and a DOE research grant DE-SC0024207.

References

[1] 2025. A Distributed Memory Object Caching System. http://memcached.org/.

[2] 2025. Redis. https://redis.io/.

[3] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-Value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 113–119. doi:10.1145/3317550.3321434

[4] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. 2020. CarMap: Fast 3D Feature Map Updates for Automobiles . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 1063–1081. https://www.usenix.org/conference/nsdi20/presentation/ahmad

[5] Akamai. 2025. EdgeKV. https://www.akamai.com/products/edgekv.

[6] AMD. 2025. AMD Pensando Networking. https://www.amd.com/en/products/accelerators/pensando.html.

[7] Ravi Shreyas Anupindi, Swaroop Kotni, and Arkaprava Basu. 2022. memwalkd : Accelerating Key-Value Stores Ssing Page Table Walkers. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 69–74. doi:10.1109/HiPC56025.2022.00021

[8] Jens Axboe. 2025. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[9] Juan Aznar-Poveda, Tobias Pockstaller, Thomas Fahringer, Stefan Pedratscher, and Zahra Najafabadi Samani. 2024. SDKV: A Smart and Distributed Key-Value Store for the Edge-Cloud Continuum *(UCC '23)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. doi:10.1145/3603166.3632126

[10] Luca Barsellotti, Faris Alhamed, Juan Jose Vegas Olmos, Francesco Paolucci, Piero Castoldi, and Filippo Cugini. 2022. Introducing Data Processing Units (DPU) at the Edge [Invited]. In *2022 International Conference on Computer Communications and Networks (ICCCN)*. 1–6. doi:10.1109/ICCCN54977.2022.9868927

[11] Mohammadreza Bayatpour, Nick Sarkauskas, Hari Subramoni, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. 2021. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer International Publishing, Cham, 18–37.

[12] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H. Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards Federated Learning at Scale: System Design. arXiv:1902.01046 [cs.LG] https://arxiv.org/abs/1902.01046

[13] Jiahong Chen, Shengzhe Wang, Zhihao Zhang, Suzhen Wu, and Bo Mao. 2023. iKnowFirst: An Efficient DPU-Assisted Compaction for LSM-Tree-Based Key-Value Stores. In *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 53–60. doi:10.1109/ASAP57973.2023.00022

[14] Shanshan Chen, Xiaoxin Tang, Hongwei Wang, Han Zhao, and Minyi Guo. 2016. Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. In *2016 IEEE Trustcom/BigDataSE/ISPA*. 1660–1667. doi:10.1109/TrustCom.2016.0255

[15] Aakanksha Chowdhery, Marco Levorato, Igor Burago, and Sabur Baidya. 2018. *Urban IoT Edge Analytics*. Springer International Publishing, Cham, 101–120. doi:10.1007/978-3-319-57639-8_6

[16] Cloudfare. 2024. Cloudflare Workers KV. https://developers.cloudflare.com/kv/.

[17] 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* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[18] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 15–28. doi:10.1145/1629575.1629578

[19] DPDK. 2023. ABI and API Deprecation. https://doc.dpdk.org/guides-20.08/rel_notes/deprecation.html.

[20] DPDK. 2023. Generic flow API (rte_flow). https://doc.dpdk.org/guides-21.08/prog_guide/rte_flow.html.

[21] DPDK. 2025. Data Plane Development Kit (DPDK). https://www.dpdk.org/.

[22] DPDK. 2025. NVIDIA MLX5 Ethernet Driver. https://doc.dpdk.org/guides/nics/mlx5.html.

[23] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[24] Cloud Native Computing Foundation. 2025. etcd. https://etcd.io/.

[25] Harshit Gupta and Umakishore Ramachandran. 2018. FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems* (Hamilton, New Zealand) *(DEBS '18)*. Association for Computing Machinery, New York, NY, USA, 148–159. doi:10.1145/3210284.3210297

[26] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 135–148.

[27] Ting He, Hana Khamfroush, Shiqiang Wang, Tom La Porta, and Sebastian Stein. 2018. It's Hard to Share: Joint Service Placement and Request Scheduling in Edge Clouds with Sharable and Non-Sharable Resources. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 365–375. doi:10.1109/ICDCS.2018.00044

[28] Intel. 2025. Infrastructure Processing Units (IPUs). https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[29] Arpan Jain, Nawras Alnaasan, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2021. Accelerating CPU-based Distributed DNN Training on Modern HPC Clusters using BlueField-2 DPUs. In *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*. 17–24. doi:10.1109/HOTI52880.2021.00017

[30] Alan Jowett. 2022. Evolving Networking with a DPU-powered Edge. https://techcommunity.microsoft.com/blog/azurestackblog/evolving-networking-with-a-dpu-powered-edge/3672898.

[31] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[32] 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* (Chicago, Illinois, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 295–306. doi:10.1145/2619239.2626299

[33] 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)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[34] Sara Karamati, Clayton Hughes, K. Scott Hemmert, Ryan E. Grant, W. Whit Schonbein, Scott Levy, Thomas M. Conte, Jeffrey Young, and Richard W. Vuduc. 2022. "Smarter" NICs For Faster Molecular Dynamics: A Case Study. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 583–594. doi:10.1109/IPDPS53621.2022.00063

[35] Arjun Kashyap, Yuke Li, Darren Ng, and Xiaoyi Lu. 2025. Understanding the Idiosyncrasies of Emerging BlueField DPUs. In *Proceedings of the 39th ACM International Conference on Supercomputing* (Salt Lake City, UT, USA) *(ICS '25)*. Association for Computing Machinery, New York, NY, USA, 15 pages. doi:10.1145/3721145.3725780

[36] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 137–152. doi:10.1145/3132747.3132756

[37] Fuliang Li, Qin Chen, Jiaxing Shen, Xingwei Wang, and Jiannong Cao. 2025. Performance Characteristics and Guidelines of Offloading Middleboxes Onto BlueField-2 DPU. *IEEE Trans. Comput.* 74, 2 (2025), 609–622. doi:10.1109/TC.2024.3500372

[38] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-Value Store Server Platform. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 476–488. doi:10.1145/2749469.2750416

[39] Yuke Li, Arjun Kashyap, Weicong Chen, Yanfei Guo, and Xiaoyi Lu. 2024. Accelerating Lossy and Lossless Compression on Emerging BlueField DPU Architectures. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 373–385. doi:10.1109/IPDPS57955.2024.00040

[40] Yuke Li, Arjun Kashyap, Yanfei Guo, and Xiaoyi Lu. 2023. Characterizing Lossy and Lossless Compression on Emerging BlueField DPU Architectures. In *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. 33–40. doi:10.1109/HOTI59126.2023.00019

[41] Yuke Li, Arjun Kashyap, Yanfei Guo, and Xiaoyi Lu. 2024. Compression Analysis for BlueField-2/-3 Data Processing Units: Lossy and Lossless Perspectives . *IEEE Micro* 44, 02 (March 2024), 8–19. doi:10.1109/MM.2023.3343636

[42] Hyeontaek Lim. 2025. mica2. https://github.com/efficient/mica2.

[43] 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)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[44] Linux. 2025. perf(1) — Linux manual page. https://www.man7.org/linux/man-pages/man1/perf.1.html.

[45] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. 2021. Performance Characteristics of the BlueField-2 SmartNIC. arXiv:2105.06619 [cs.NI] https://arxiv.org/abs/2105.06619

[46] Jiani Liu, Ju Ren, Yongmin Zhang, Sheng Yue, and Yaoxue Zhang. 2024. SESAME: A Resource Expansion and Sharing Scheme for Multiple Edge Services Providers. *IEEE/ACM Transactions on Networking* 32, 4 (2024), 3189–3204. doi:10.1109/TNET.2024.3377908

[47] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High Performance RDMA-based MPI Implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing* (San Francisco, CA, USA) *(ICS '03)*. Association for Computing Machinery, New York, NY, USA, 295–304. doi:10.1145/782814.782855

[48] 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* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 318–333. doi:10.1145/3341302.3342079

[49] Antonios Makris, Evangelos Psomakelis, Theodoros Theodoropoulos, and Konstantinos Tserpes. 2022. Towards a Distributed Storage Framework for Edge Computing Infrastructures. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge* (Minneapolis, MN, USA) *(FRAME '22)*. Association for Computing Machinery, New York, NY, USA, 9–14. doi:10.1145/3526059.3533617

[50] Marvell. 2025. Marvell Data Processing Units (DPUs). https://www.marvell.com/products/data-processing-units.html.

[51] Chris Mellor. 2023. Nebulon Unveils Compact DPU for Edge Servers. https://blocksandfiles.com/2023/11/22/nebulon-developing-dpu-to-offload-fleets-of-edge-servers/.

[52] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. 2018. Toward Session Consistency for the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotedge18/presentation/mortazavi

[53] Mohammadreza Najafi, Kaiwen Zhang, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2017. Hardware Acceleration Landscape for Distributed Real-Time Analytics: Virtues and Limitations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 1938–1948. doi:10.1109/ICDCS.2017.194

[54] Zhen Ni, Cuidi Wei, Timothy Wood, and Nakjung Choi. 2021. A SmartNIC-based Load Balancing and Auto Scaling Framework for Middlebox Edge Server. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 21–27. doi:10.1109/NFV-SDN53031.2021.9665167

[55] Joseph Noor, Mani Srivastava, and Ravi Netravali. 2021. Portkey: Adaptive Key-Value Placement over Dynamic Edge Networks. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 197–213. doi:10.1145/3472883.3487004

[56] NVIDIA. 2021. Accelerating Edge Computing with a Smarter Network. https://developer.nvidia.com/blog/accelerating-edge-computing-with-a-smarter-network/.

[57] NVIDIA. 2023. Transform the Data Center for the AI Era with NVIDIA DPUs and NVIDIA DOCA. https://developer.nvidia.com/blog/transform-the-data-center-for-the-ai-era-with-nvidia-dpus-and-nvidia-doca/.

[58] NVIDIA. 2025. DOCA DMA. https://docs.nvidia.com/doca/sdk/doca+dma/index.html.

[59] NVIDIA. 2025. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[60] NVIDIA. 2025. NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[61] NVIDIA. 2025. NVIDIA BlueField DPU Modes of Operation. https://docs.nvidia.com/doca/sdk/bluefield+modes+of+operation/index.html.

[62] NVIDIA. 2025. NVIDIA DOCA Software Framework. https://developer.nvidia.com/networking/doca.

[63] István Pelle, Márk Szalay, János Czentye, Balázs Sonkoly, and László Toka. 2022. Cost and Latency Optimized Edge Computing Platform. *Electronics* 11, 4 (2022). doi:10.3390/electronics11040561

[64] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The Benefits of General-Purpose on-NIC Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1130–1147. doi:10.1145/3503222.3507711

[65] Arun Ravindran and Anjus George. 2018. An Edge Datastore Architecture For Latency-Critical Distributed Machine Vision Applications. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotedge18/presentation/ravindran

[66] Mauro Salazar, Matthew Tsao, Izabel Aguiar, Maximilian Schiffer, and Marco Pavone. 2019. A Congestion-aware Routing Scheme for Autonomous Mobility-on-Demand Systems. In *2019 18th European Control Conference (ECC)*. 3040–3046. doi:10.23919/ECC.2019.8795897

[67] Md. Maruf Hossain Shuvo, Syed Kamrul Islam, Jianlin Cheng, and Bashir I. Morshed. 2023. Efficient Acceleration of Deep Learning Inference on Resource-Constrained Edge Devices: A Review. *Proc. IEEE* 111, 1 (2023), 42–91. doi:10.1109/JPROC.2022.3226481

[68] Karim Sonbol, Öznur Özkasap, Ibrahim Al Oqily, and Moayad Aloqaily. 2020. EdgeKV: Distributed Key-Value Store for the Network Edge. In *2020 IEEE Symposium on Computers and Communications (ISCC)*. 1–6. doi:10.1109/ISCC50000.2020.9219667

[69] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 347–353. https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi

[70] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. 2022. SKV: A SmartNIC-Offloaded Distributed Key-Value Store. In *IEEE International Conference on Cluster Computing, CLUSTER 2022, Heidelberg, Germany, September 5-8, 2022*. IEEE, 1–11. doi:10.1109/CLUSTER51413.2022.00016

[71] Lasse Thostrup, Daniel Failing, Tobias Ziegler, and Carsten Binnig. 2022. A DBMS-centric Evaluation of BlueField DPUs on Fast Networks. In *13th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*.

[72] Khikmatullo Tulkinbekov and Deok-Hwan Kim. 2020. CaseDB: Lightweight Key-Value Store for Edge Computing Environment. *IEEE Access* 8 (2020), 149775–149786. doi:10.1109/ACCESS.2020.3016680

[73] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. 2021. LaSS: Running Latency Sensitive Serverless Computations at the Edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (Virtual Event, Sweden) *(HPDC '21)*. Association for Computing Machinery, New York, NY, USA, 239–251. doi:10.1145/3431379.3460646

[74] Qing Wang, Youyou Lu, Jing Wang, and Jiwu Shu. 2023. Replicating Persistent Memory Key-Value Stores with Efficient RDMA Abstraction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 441–459. https://www.usenix.org/conference/osdi23/presentation/wang-qing

[75] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, and Zili Shao. 2010. Optimal Task Scheduling by Removing Inter-Core Communication Overhead for Streaming Applications on MPSoC. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 195–204. doi:10.1109/RTAS.2010.19

[76] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 987–1004. https://www.usenix.org/conference/osdi23/presentation/wei-smartnic

[77] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. doi:10.1145/2815400.2815419

[78] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '15)*. Association for Computing Machinery, New York, NY, USA, Article 6, 11 pages. doi:10.1145/2757667.2757684

[79] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3, Article 17 (Aug. 2021), 35 pages. doi:10.1145/3468521

[80] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan R. K. Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. 2023. Rambda: RDMA-driven Acceleration Framework for Memory-intensive µs-scale Datacenter Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 499–515. doi:10.1109/HPCA56546.2023.10071127

[81] Yongmin Zhang, Xiaolong Lan, Ju Ren, and Lin Cai. 2020. Efficient Computing Resource Sharing for Mobile Edge-Cloud Computing Networks. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1227–1240. doi:10.1109/TNET.2020.2979807

[82] Xingyu Zhou, Robert Canady, Shunxing Bao, and Aniruddha Gokhale. 2020. Cost-effective Hardware Accelerator Recommendation for Edge Computing. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association. https://www.usenix.org/conference/hotedge20/presentation/zhou-xingyu

[83] Guangxu Zhu, Yong Wang, and Kaibin Huang. 2020. Broadband Analog Aggregation for Low-Latency Federated Edge Learning. *IEEE Transactions on Wireless Communications* 19, 1 (2020), 491–506. doi:10.1109/TWC.2019.2946245