DPC: DPU-accelerated High-Performance File System Client

Kan Zhong
Zhiwang Yu
kzhong@cqu.edu.cn
zwyu@stu.cqu.edu.cn
School of Big Data and Software Engineering
Chongqing University
Chongqing, China

Qiao Li
liqiao@xmu.edu.cn
School of Informatics
Xiamen University
Xiamen, China

Xianqiang Luo*
luoxq21@mails.tsinghua.edu.cn
Department of Computer Science and Technology
Tsinghua University
Beijing, China

Linbo Long
longlb@cqupt.edu.cn
School of Computer Science and Technology
Chongqing University of Posts and Telecommunications
Chongqing, China

Yujuan Tan
Ao Ren
tanyujuan@cqu.edu.cn
ren.ao@cqu.edu.cn
College of Computer Science
Chongqing University
Chongqing, China

Duo Liu
liuduo@cqu.edu.cn
School of Big Data and Software Engineering
Chongqing University
Chongqing, China

ABSTRACT

To achieve efficient file access to the file system backend, file system clients employ various intricate optimization techniques, such as local data/metadata caching and direct data access. However, these techniques impose a significant load on the host CPU, posing substantial challenges to the valuable CPU resources.

In this paper, we propose ***DPC***, a **D**PU-accelerated High-**P**erformance **C**lient designed for both distributed and standalone file systems. DPC offloads complex file-semantic operations, such as file or directory delegation, cache management, and erasure code computation, from the host CPU to the DPU, thus freeing up the host CPU cycles. First, to offer low-latency and high-performance file operations, we introduce `nvme-fs`, which allows applications to directly interact with DPC through native file semantics by augmenting the NVMe protocol. Second, a hybrid caching mechanism with separation of the data plane and control plane is proposed in DPC to accelerate file accesses. Finally, a key-value based file system, namely KVFS, is proposed in DPC to provide local standalone file service, thus replacing the under-utilized local disks with disaggregated storage.

Experimental results show that DPC can achieve extremely low (as low as 20μsec) host-DPU transmission latency. The local standalone file service provided by DPC outperforms Ext4 and saves more than 80% CPU usage in high concurrency scenarios. Besides, with DPC, distributed file systems can reduce the host CPU usage by 90% while maintaining high performance.

*Corresponding author.

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673123

CCS CONCEPTS

• **Software and its engineering** → **File systems management**; • **Information systems** → *Cloud based storage*; *Distributed storage*; • **Networks** → Network File System (NFS) protocol; Programmable networks; Network adapters.

KEYWORDS

Distributed file system, DPU, Offloading, NVMe

ACM Reference Format:
Kan Zhong, Zhiwang Yu, Qiao Li, Xianqiang Luo, Linbo Long, Yujuan Tan, Ao Ren, and Duo Liu. 2024. DPC: DPU-accelerated High-Performance File System Client. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673123

1 INTRODUCTION

The file system client (a.k.a fs-client) installed on the application servers enables file access to the backend file systems by providing standard POSIX-compliant or customized interfaces. Existing fs-clients are primarily designed for distributed file systems (DFS) [1, 8, 14, 16, 17, 25, 39] and face several issues. First, to make shared DFS performed as local standalone file systems (i.e., Ext4), many file semantic operations, such as erasure code (EC) calculation, I/O request forwarding, cache management, and lock delegation, are directly executed on the file system client. For example, LustreFS [25] enables client-side data compression and encryption to reduce network traffic and protect cache disks against theft or loss [11]. These client-side optimizations improve the efficiency of file operations but add up to 30% "datacenter tax" [19] in CPU usage. Second, to make file systems completely POSIX-compliant, some fs-clients choose to deploy a feature-rich module in the kernel space to interact with the virtual file system (VFS) layer, like the fs-client in LustreFS. In contrast, other file systems choose to intercept the VFS file operations and pass these operations to user-space fs-client for further processing via the FUSE library [36], like WeKaFS [40]. The

former sacrifices flexibility and is not permitted in some environments for security reasons. At the same time, the latter suffers from both latency and bandwidth bottlenecks introduced by the narrow kernel-userspace interaction channel in the FUSE framework.

To address these challenges, offloading disaggregated file system stacks has become increasingly prevalent. Data processing units (DPUs) or SmartNICs, such as BlueField, Fungible-DPU, Nitro, and Catapult, as referenced in [3, 9, 13, 18], are well-suited options for offloading tasks, due to their position in the data path of disaggregated file operations and their combined capabilities in computation and networking. To this end, IBM proposes DPFS [15], which offloads the fs-client to a client-side DPU and utilizes the Linux `virtio-fs` software stack to transfer file operations to the DPU. LineFS [20] offloads CPU-intensive DFS tasks, like replication and compression, to a server-side SmartNIC. However, DPFS introduces a substantial performance penalty due to the involvement of multiple DMA operations within the `virtio-fs` data transfer protocol. On the other hand, LineFS has been meticulously crafted with a focus on PM-optimized DFS tasks and does not consider fs-client's offload requirements.

In this paper, we propose DPC, a DPU-accelerated high-performance client that provides both standalone file service as local file system and shared distributed file service as DFS. DPC is designed for diskless architecture, where I/O operations are separated from the CPU and offloaded to dedicated hardware like DPU and SmartNIC. In DPC, we first discard the slow `virtio-fs` designed for virtualization and propose `nvme-fs`, a high-performance NVMe-based file protocol for the DPU-offloaded file system. `nvme-fs` enables the VFS to interact with DPU-offloaded file system stacks through native file semantics by augmenting the NVMe protocol. Second, due to the limited memory capacity of DPU, DPC adopts a hybrid cache management scheme that uses host memory as the cache space while putting the cache control plane into the DPU. The hybrid scheme not only largely reduces the PCIe traffic but also offloads the cache management task to DPU, enabling the adoption of a more flexible and intelligent caching algorithm. Third, most application servers use local file systems such as Ext4, which are built based on local disks to store local and configuration files. However, these local-disk architectures can result in low utilization and imbalanced allocation of storage resources. Therefore, DPC provides KVFS, a lightweight and POSIX-compliant standalone file service that uses disaggregated storage to replace the under-utilized local disks, enabling diskless architecture for application servers. KVFS runs in DPC and converts the local file operations to remote KV operations. In this way, DPC provides high-performance file services for diskless architecture, freeing the CPU from complex file stacks and allowing the CPU to focus on the computing and processing of application workloads.

Experimental results show that DPC can outperform the state-of-art DPFS by 2–3 times in terms of both IOPS and latency. DPC can provide a high-salable local standalone file service that outperforms Ext4 and saves more than 80% host-side CPU usage in high concurrency scenarios. Besides, with DPC, distributed file systems can achieve 90% host-side CPU usage reduction while maintaining high performance.

In summary, we make the following contributions.

- To free the CPU from the complex file stacks, we propose DPC, a DPU-accelerated file system client that provides both standalone file service and shared distributed file service.
- We propose `nvme-fs` for DPC, a high-performance NVMe-based file protocol for DPU-offloaded file system stacks, permitting the VFS layer to communicate with DPC through native file semantics.
- We propose a hybrid cache, which offloads the control plane to DPU while leaving the cache data plane to reside on the host side to fully utilize host-side memory space.
- We design KVFS for DPC, which allows DPC to provide standalone file service as local file systems, thus eliminating the need for local physical disks.
- We evaluate and analyze the effectiveness of DPC in various scenarios and compare it to other works.

The rest of this paper is organized as follows. In Section 2, we introduce the background and motivation. We present DPC's design and implementation details in Section 3. Section 4 evaluates the effectiveness of DPC. Section 5 presents the related works. Finally, we give the conclusion of this work in Section 6.

2 BACKGROUND AND MOTIVATION

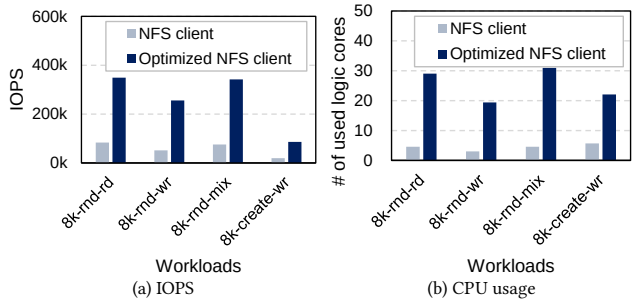
In this section, we first present the background on DFS and DPU offloading. Then, we give the motivation to propose DPC.

2.1 Distributed File System

File systems are vital for users to store and access data. DFS, like GFS [14], HDFS [16], Lustre [25], Ceph [39], DAOS [8], Pangu [1] and OceanStor Pacific [17] offer excellent system scalability and reliability by separating metadata management from file data storage. Typically, these distributed file systems consist of three components: metadata server (MDS), data server, and fs-client. MDS manages the metadata of the file system, including directory structures (i.e., dentry tables), file attributes (i.e., inode tables), privileges, locks, and file data layouts. The data server is responsible for file and metadata persistence and provides a set of interfaces for data manipulation. The fs-client is a driver installed in the application server and initiates file requests to the backend. For a POSIX-compliant DFS, file requests are first sent to the Linux VFS layer and then passed to the fs-client. Usually, when accessing a file, fs-client will first communicate with MDS to retrieve the file metadata and then request the file in the data server based on the file metadata. In this path, several optimizations described as follows have been adopted in the fs-client to accelerate the I/O path.

Client-side I/O forwarding. Since file metadata is evenly distributed among all MDSes, fs-client has to send the metadata requests to its entry MDS, which acts as a metadata proxy and forwards the requests to the home MDS that stores the metadata. The I/O forwarding process not only increases the latency but also consumes more CPU cycles of the entry MDS. To solve this issue, fs-client can cache the metadata distribution information (a.k.a metadata view) and directly route the metadata requests to their home MDSes.

Client-side EC calculation. Normally, EC is calculated by the file's home MDS and then distributed to multiple data servers. To



(a) IOPS (b) CPU usage

Figure 1: Optimization of file stacks at the fs-client improves the IOPS by around 4 times, but also incurs around 4–6 times more CPU core consumption. Note that the `mix` workload is comprised of 70% random read and 30% random write.

reduce the latency, the fs-client can calculate the EC and write the strip units directly to data servers with direct I/O.

Direct I/O (DIO). Small I/O with its metadata can be packed into a single message and sent to MDS. Subsequently, MDS consolidates multiple small I/Os into a single large I/O and transmits it to the data server to minimize network round trips. Nonetheless, this data path is not optimal for large I/Os. With client-side DIO, large I/Os can be directly written to data nodes and their metadata is updated lazily in batches. Thus, the network bandwidth can be fully utilized to improve the throughput of large I/Os. In this process, EC or replication is handled by the fs-client.

2.2 Data Processing Unit

SmartNIC is a powerful network adapter that usually has programmable logic to accelerate the processing of network packets [21, 31, 41]. Most of the existing SmartNICs adopt the *on-path* architecture, which directly exposes the network processing (NP) cores for handling network packets to the system [24, 38]. Although the on-path processing is highly efficient, developers must carefully handle the limited on-chip cache and context switch with low-level programming interfaces, making the programming of NP cores extremely hard. DPU, such as NVIDIA BlueField [9] and Fungible F1 [18], usually use *off-path* architecture and can be regarded as an evolution of SmartNIC. In DPU, a general-propose multi-core CPU with DRAM is attached next to the NP cores via the PCIe switch. With this separation, the CPU is independent of the network processing path, and thus a full-fledged OS can be installed to simplify the programming. Given that DPU is set on the data path of accessing disaggregated storage, its network capabilities are pretty powerful. Most DPUs support InfiniBand and RoCE (RDMA over Converged Ethernet) protocol to enable high-speed (up to 400Gpbs) Remote Direct Memory Access (RDMA). However, a high-speed network requires more CPU resources to saturate the bandwidth of RDMA-capable NIC. Therefore, the trend becomes more popular to offload I/O intensive tasks into dedicated DPU to ease the burden of host-side CPU.

In this paper, we design DPC based on the off-path DPU to offload the complex file operations. The detailed configuration of DPU can be found in Section 4.

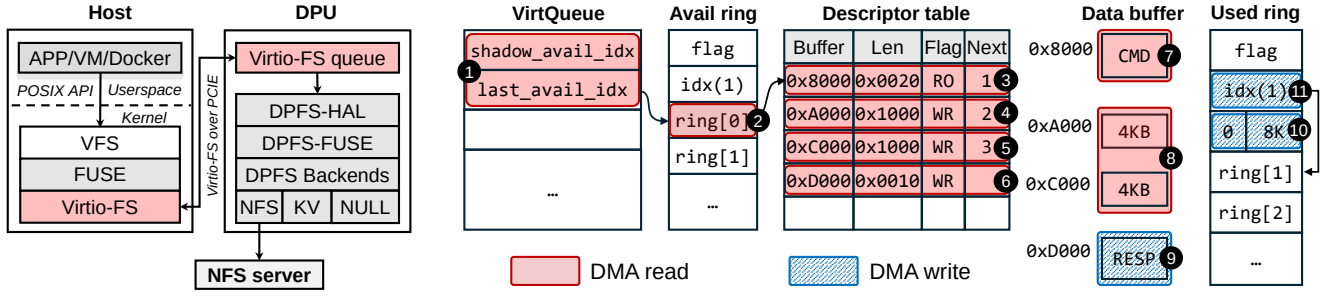
2.3 Motivation

We propose DPC based on the following motivations.

M1. High resource usage of optimized fs-client. The client for DFS enables file access to the disaggregated file system backend. To improve the performance of DFS to approach the local file system, many file semantics operations and data processing tasks are executed directly in the fs-client. For example, file metadata view can be maintained in fs-client to avoid forwarding, erasure code can be calculated in fs-client and then directly written to the data node to reduce the write latency, and file locks or delegations can be cached in fs-client to improve the lock acquire performance. All these optimizations can largely improve the performance of DFS but also make the fs-client more complex, consuming significant amounts of memory and CPU resources. Figure 1 compares the IOPS and CPU usage between a standard NFS client and an optimized NFS client that enables EC calculation, I/O forwarding, file delegations and DIO. The optimized NFS client outperforms the standard NFS client by more than 4 times in terms of IOPS, but it brings around 6 times more CPU core consumption. Therefore, there is a new trend of offloading file system stacks to DPUs or SmartNICs to provide high-scalability and high-performance file system services.

M2. Inefficient file stack offloading path. DPFS is the state-of-art work for fs-client offloading. In DPFS, as shown in Figure 2(a), file operation requests are first sent to the VFS layer through system calls. Then, the requests are forwarded to the FUSE layer, where the requests will be converted into FUSE messages. Finally, the FUSE messages are passed to the DPU via the `virtio-fs`. On the DPU side, a DPFS-HAL thread will retrieve the messages from the `virtio-fs` queue and then pass them to DPFS-FUSE. The DPFS-FUSE extracts the original FUSE requests and forwards them to the file system backend. However, using the PCIe memory-mapped `virtio` queues and the FUSE file I/O command set brings several performance problems. First, missed requests in the page cache are transformed within the FUSE layer and deposited into the FUSE queue for processing. Unfortunately, the structure of the FUSE queue is overburdened, leading to a prolonged request response time. Second, the read and write operations of `virtio-fs` involve a significant amount of DMA operations that greatly damage the system performance.

We take 8KB data write as an example, as shown in Figure 2(b), the number of DMA operations involved in `virtio-fs` reaches up to unbearable 11. When the request initiated by applications is routed to the queue of `virtio-fs`, a DPFS-HAL thread in DPU starts to process the request. Concretely, the thread first gets the *last_avail_idx* which indicates the position where it starts processing the request in the *Avail ring* (❶). Then, based on the value of *last_avail_idx* (0 in this case), the thread reads the corresponding entry of the ring array (❷) in the *Avail ring* to locate the head of the data buffer chain recorded in the *Descriptor table*. Following this, the thread starts to read the entries of the data buffer chain one by one according to the *Next* field, where each entry locates the address of the corresponding buffers (❸–❻). Subsequently, the thread starts to process data buffers by reading the command (❼) and data (❽) written by applications. After the DPU completes the command processing, it will return the host with a response (❾). In response to the host, the index (i.e., 0) of the buffer chain's head entry in the *Descriptor table* and the length (i.e., 8K) of the read data processed in this case will be added to the *Used ring* (❿). Finally, the thread increases the value of the *idx* in the *Used ring* by one



(a) Architecture of DPFS

(b) 8K data write with `virtio-fs` in DPFS. The I/O transmission involves 11 DMAs

Figure 2: The file operation offloading path in DPFS. Due to the inefficiency of `virtio-fs`, data transmission in DPFS involves multiple DMA operations, which will negatively impact the latency and IOPS.

(⓫) to indicate the next store position of the head and data length of the newly processed data buffer chain. What makes the situation worse is that the current kernel implementations of DPFS do not support multiple queues. Hence, DPFS can only employ a single DPFS-HAL thread to process a single `virtio-fs` queue, severely limiting the system throughput.

M3. Low utilization of local disks. Besides the DFS, application servers are usually equipped with several local disks to store local and configuration files. For example, virtualization cloud vendors use local disks to store container or virtual machine images. However, this approach would lead to low and imbalanced utilization of local disks. Prior research [6, 33] has shown that the disk utilization of cloud environments is lower than 20% for most of the time. Therefore, replacing the under-utilized local disks with disaggregated storage and enabling diskless architecture for application servers becomes a critical mission in reducing the total cost of ownership for cloud vendors.

Based on these motivations, we propose DPC for diskless architecture to solve the "datacenter tax" in I/O operations by offloading the non-CPU-friendly file stacks to DPU. DPC provides both standalone and shared distributed file services.

3 DESIGN AND IMPLEMENTATION

In this section, we first present the design overview of DPC. Then, we detail the design of `nvme-fs`, an NVMe-based high-performance file protocol for DPU-offloaded file stacks. Following that, the hybrid cache mechanism that separates control plane from data plane will be represented. Finally, we describe the KVFS, which allows DPC to provide standalone file service as local file systems, eliminating the need for local physical disks.

3.1 Overview

The overall architecture of our proposed DPC is shown in Figure 3. DPC is deployed in the DPU and provides both standalone file service and shared DFS service to applications through the high-performance `nvme-fs` protocol. There are three key designs in the proposed DPC: 1) `nvme-fs`, a file-native DPU offloading protocol based on the NVMe protocol; 2) hybrid cache that separates the cache control plane from cache data plane and offloads the control plane to DPU; 3) KVFS, a KV-based file system that runs in DPC and provides standalone file services like local Ext4.

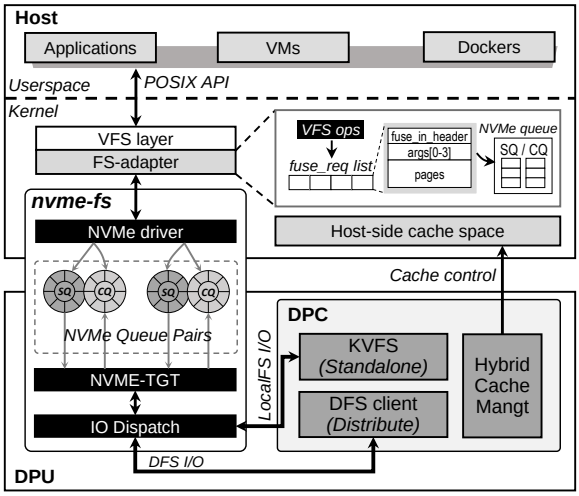


Figure 3: Architecture overview of the proposed DPC.

As shown, the file read/write requests will first be passed to the VFS layer through syscalls. In the host kernel, a lightweight adapter, namely *fs-adapter*, instead of FUSE, is employed to interact with the VFS layer. *fs-adapter* is responsible for reading/writing the hybrid cache space and converting the file requests into messages that are used for `nvme-fs`. Besides, *fs-adapter* eliminates the bloated queues of FUSE and provides more efficient queue operations. For file read requests, *fs-adapter* will first search the hybrid cache space and then issue the requests to DPU if the cache is not hit. For write requests, the data will be cached in the hybrid cache space directly if the `DIRECT_IO` flag is not specified. On the DPU side, an I/O dispatch module is employed to dispatch the file requests to the KVFS or the DFS client stacks, depending on the request type.

The proposed `nvme-fs` (details in Section3.2) achieves efficient interaction between the host and DPU by augmenting the NVMe protocol. Therefore, as shown in Figure 3, the NVME-INI driver and NVME-TGT driver are used to manipulate NVME queues for the host and DPU, respectively. `nvme-fs` significantly outperforms `virtio-fs` due to the multiple queues and fewer DMA operations in NVME protocol.

Due to the limited memory capacity of DPU, DPC adopts a hybrid cache management scheme (details in Section3.3) that uses host memory as the cache space while putting the cache control plane into the DPU. The hybrid scheme not only largely reduces the PCIe traffic but also frees the host CPU from cache management. In the

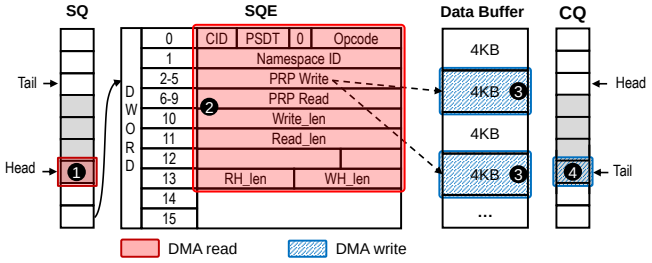


Figure 4: 8K data write with nvme-fs in DPC. The I/O transmission only involves 4 DMAs.

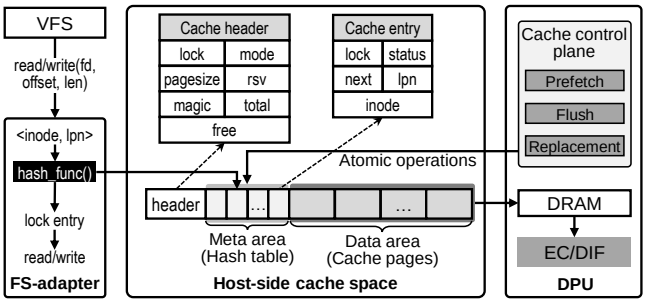


Figure 5: The proposed hybrid cache scheme. The cache control plane is offloaded to DPU while the cache data plane still resides in the host.

offloaded cache control plane, more flexible and intelligent caching algorithms can be adopted to improve the caching performance. Besides, compared to cache full offloading, double caching (i.e., host page cache and DPU-based cache) can be eliminated by using one hybrid cache layer.

The standalone file service and distributed file service in the DPC are offloaded into the DPU. To provide a lightweight and POSIX-compliant standalone file service, we implement KVFS (details in Section3.4) that uses disaggregated storage to replace the under-utilized local disks. In KVFS, the local file operations are converted into KV operations to the disaggregated KV store, which is used to store all the file data and metadata. With KVFS, DPC can provide POSIX-compliant standalone file services like the local Ext4 file system does, thus replacing the under-utilized local disks with disaggregated storage and reducing the cost of ownership for cloud vendors. Moreover, KVFS frees the host CPU from the local file system stacks, thus reserving more CPU resources for computational tasks.

3.2 *nvme-fs*

As mentioned before, `virtio-fs` read and write data entail a large number of DMA operations. For an 8KB-data write with `virtio-fs` in DPFS, the I/O transmission involves 11 DMAs, which drastically damages system performance. Therefore, we propose `nvme-fs` to achieve a high-performance file protocol for DPU-offloaded file stacks while permitting the VFS layer to communicate with DPC through native file semantics. The `nvme-fs` is based on the NVMe queue pairs (i.e., Submission Queue (SQ) and Completion Queue (CQ)) to implement the communication between the host and client in a producer-consumer mode. For a submission queue, commands are produced by NVME-INI driver at the tail of the SQ and consumed by NVME-TGT driver at the head of SQ. In contrast, for a completion queue, commands are produced by the NVME-TGT driver at the tail of the CQ and consumed by the NVME-INI driver at the head of the CQ. Figure 4 details the I/O transmission path in `nvme-fs`.

When the 8KB data write request is converted into a submission command and put into the tail of the submission queue by the NVMe-INI driver, the host notifies the DPU to process the write request. In this process, the physical address of the user data buffer is directly attached to the submission command for DMA transfer to ensure zero-copy of user data. As shown in Figure 4, for an 8KB data write with `nvme-fs` in DPC, the I/O transmission only involves 4 DMAs. First, the NVME-TGT driver retrieves the submission command entry (SQE) in SQ(❶). Then, the DPU reads the SQE and locates the data buffer indicated by `PRP Write` field(❷). Once the DPU finishes reading the data written by applications (❸), it submits an entry to the tail of the completion queue (CQ) in response to the host request(❹).

To further improve the communication efficiency between the host and DPC, we implement bidirectional communication semantics within a single command by modifying the structure of the SQE. First, we set the value of the `Opcode` (the first byte of the `Dword0`) as '0xA3', which is reserved for vendors to achieve customized functions. In detail, the lowest two bits of the `Opcode` are '11b', indicating we choose the bidirectional data transfer mode. The `Opcode`'s $2nd$ bit to the $6th$ bit is set as '01000b' to represent the corresponding function, and the highest bit is '1b', which indicates the function is customized. Then, we use the $14th$ and $15th$ bit residing in `PSDT` of the `Dword0` to indicate which structures we use to transfer the associated data or metadata for write and read requests, respectively. '0' suggests Physical Region Page (PRP), and '1' represents Scatter Gather List (SGL). In our design, we use PRP as the default structure for data transfer. Hence, the $14th$ and $15th$ bits of the `Dword0` are all set as '0b'. Following that, the `PRP Write` and `PRP Read` entries passed in `Dword2-5` and `Dword6-9` are assigned to locate the write buffer and read buffer for bidirectional data transfer, respectively. Additionally, the `Write_len` in `Dword10` and `Read_len` in `Dword11` respectively indicate the bytes of data to be written and read, while `RH_len` and `WH_len` in `Dword13` respectively indicate the number of bytes taken up by the write and read headers. Finally, we utilize the reserved $10th$ bit of the `Dword0` to store the request's type, which will be used by the `IO_Dispatch` module. '0' represents the standalone file request, which will be dispatched to the KVFS, while '1' represents the distributed file requests, which will be dispatched to the DFS client for processing.

3.3 Hybrid File Data Cache

File data cache is employed in file systems to accelerate file access. In the DPU-accelerated file system client, the cache can be entirely offloaded to the DPU to reduce CPU usage on the host. But this will bring three problems: 1) Once a cache hit occurs, the requested file data still needs to be transferred over the PCIe interface, involving multiple DMA operations between the host and DPU, thus increasing the PCIe traffic; 2) The duplication of the page cache maintained by the host-side VFS layer and the cache in the DPU leads to a double caching issue; 3) Due to the limited memory capacity of DPU, completely offloading the cache to DPU will restrict the I/O performance and application scalability.

Therefore, we propose a hybrid cache scheme that offloads the cache control plane to the DPU while leaving the cache data plane to reside on the host side. As shown in Figure 5, the proposed hybrid cache achieves high performance and scalability by using the host memory as the cache space while offloading the cache management tasks into the DPU. We argue that the proposed hybrid cache is beneficial to the file system performance in at least three aspects: 1) offloading the CPU-intensive cache management tasks to DPU to free host-side CPU cycles, 2) enabling the flexibility of customized cache replacement and prefetching algorithms to improve the cache hit rate, and 3) eliminating the memory DMA operations between host and DPU.

Cache Layout. When the file system is mounted, a cache space is reserved on the host side as a DMA-accessible memory area. Then, the address and length of the cache are sent to the DPU for initialization. The cache is a continuous block of memory space, and its layout is shown in Figure 5. The cache `header` stores the overall information of the cache. In detail, `pagesize` specifies the page size, usually 4KB; `mode` denotes the cache mode, with 0 indicating read cache and 1 indicating write cache; `total` indicates the total page count in the cache while `free` specifies the number of available pages. The meta area follows the header in a contiguous manner and comprises several cache entries, with each cache entry documenting the details of one page. For instance, in a cache entry, the `lock` displays the locking status of a page: 0 reflects no lock, 1 indicates a write lock, 2 indicates a read lock, and 3 signifies invalid. The `status` responds to whether the corresponding page has undergone modifications: 0 indicates the cache entry is free, 1 indicates that the corresponding page is clean, 2 indicates that the corresponding page is dirty, and 3 indicates that the page is invalid. The `next` is used as a pointer specifying the next cache entry in the same hash bucket (as described below). The `lpn` denotes the logical page number of a file, and the `inode` denotes the inode number of the file. The data area that consists of cache pages immediately follows the meta area. The number of cache entries and pages are the same and correspond to each other. Therefore, finding the position of the cache entry is equivalent to locating the cache page.

To quickly find the cache entry, the meta area serves as a hash table, with hash functions used to map `<indoe, lpn>` to the cache entry. The hash table (i.e., meta area) is logically divided into buckets, and each bucket has the same number of cache entries. Cache entries in a bucket are linked in a list by the `next` pointer. When searching a page in the cache, the `<indoe, lpn>` is first used by the hash function to find the corresponding bucket, and then search the page entry list by comparing the `inode` and `lpn`. Similarly, when inserting a page into the cache, after finding the corresponding bucket, a free cache entry is selected from the cache entry list. If there is no free page entry, page replacement will be performed to reclaim cache space.

Data Consistency. The key challenge of a hybrid cache is maintaining consistency between the host and DPU. It may lead to data in an inconsistent state when DPU is performing flush operations for the cache while the corresponding pages are being modified by the host. To address this problem, we utilize read-and-write locks encapsulated by the PCIe atomic mechanism to achieve concurrency control of the meta area. To ensure the consistency of the file data area, a page can be accessed only after the corresponding cache entry has been locked. Based on this, the front-end write process is as follows. Host first finds a free cache entry by calculating the hash function according to `<inode, lpn>` of the file page. After allocating a free cache entry, the host locks the cache entry atomically. If it fails to allocate and lock, the host notifies the DPU to perform cache replacement. Then, based on the position of the cache entry, the host calculates the page address in the data area and writes the data to the cache page. After that, the host atomically releases the write lock on the cache entry and sets its dirty status to complete the write operation.

For cache flushing, the DPU periodically scans the hash table of the meta area to safely flush the selected dirty pages by adding the read locks for them. To do this, DPU temporarily pulls the data to its DRAM by DMA transmission and performs relevant computing operations (e.g., compression, DIF, EC, etc.) as needed (this step can be accelerated by hardware). Then, DPU writes these data to the disaggregated storage. After completing flushing, DPU releases the read locks of cache entries in the meta area atomically and updates their status to clean or free. The front-end read process is similar to the write process.

3.4 Local Standalone File Service

Due to the extremely low utilization of local disks in application servers, there is a growing trend to use disaggregated storage to build diskless architectures. The key challenge for diskless architecture is how to provide standalone file services as the local Ext4 does. To this end, we designed KVFS for DPC. KVFS runs in DPU and is responsible for converting file operations from the VFS layer to disaggregated KV operations. Cooperating with the proposed `nvme-fs`, KVFS offloads the file semantics-related operations to DPU and provides a fully POSIX-compliant file service. To achieve this, four different types of KV are used in KVFS to represent the file and direct structure: inode KV, file attribute KV, small file data KV, and big data KV.

Inode KV: `[key: p_ino+name; value: ino]`. Inode KV is used to store the mapping from the file or directory name to its corresponding inode number. Thus, the key is comprised of the parent directory's inode number (`p_ino`) and file or directory name (`name`). The value is a 64-bit integer that represents the inode number of the file or directory. In KVFS, we have limited the length of the file or directory name to 1024 bytes. Thus, the maximum length of the key is 1088 bytes. For inode KV, the `p_ino` is a key prefix, and it is used for directory list operations — a prefix-based scan can return all the inode numbers belonging to a directory specified by the `p_ino`.

Attribute KV: `[key: ino; value: attribute]`. Attribute KV maps the inode number to the attribute of a file or directory. The `attribute` value is a 256-byte data structure that describes the file or directory's privilege, size, ownership, creation time, and so on.

Small file KV: `[key: ino; value: file data]`. For small files (less than 8KB), we store their data in traditional key-value pairs. When updating the file data, we rewrite the entire KV. When the file size grows bigger than 8KB, KVFS deletes the small file KV and creates a big file KV.

Big file KV: `[key: ino; value: file data]`. For big files (larger than 8KB), rewriting the whole KV upon each file update

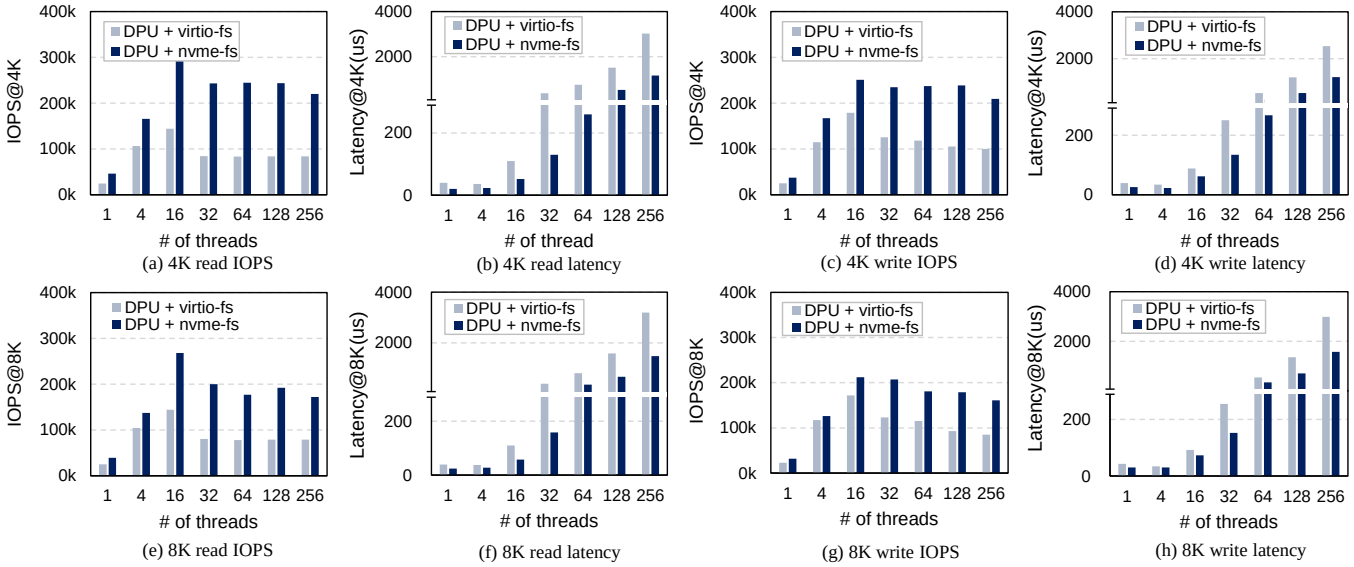


Figure 6: The raw host-DPU data transmission IOPS and latency comparison between `virtio-fs` and the proposed `nvme-fs` under different concurrency.

is inefficient and would introduce large data amplification for the storage system. Therefore, we have introduced an in-place update capability for large file KVs so that updates to large files are written in place to large file KVs at a granularity of 8K. To achieve this, big file KV uses the file object designed for DFS, in which each file is associated with a file object. The file object uses an index structure to map the underlying discrete physical storage blocks into its own contiguous file space.

In KVFS, the root directory has a unique inode number 0. Thus, path resolution is done by recursively fetching the inode KVs from the root to the target inode using `p_ino+name` as the key. KVFS is implemented under VFS and is compatible with VFS, thus the inode cache and dentry cache can also be used to speed up the file or directory lookups.

Note that this paper does not focus on the design of disaggregated storage, so we do not introduce the design details of the disaggregated KV store and the DFS backend. Instead, we focus on the fs-client offloading at the client side, so we show how the DFS client can be offloaded to DPU and how to provide local standalone file service by leveraging the disaggregated KV store.

4 EVALUATION

In this section, we evaluate the performance and efficiency of DPC. First, we present the comparison of raw data transmission performance between `virtio-fs` and `nvme-fs`. Then, we illustrate the performance comparison between the local file system(i.e., Ext4) and the proposed KVFS. Finally, we show the contributions of DPC to DFS in both performance and host-side CPU usage reduction. Table 1 shows our evaluation setups. We use *vdbench* [26] and *fio* [5] in our evaluation to generate I/O workloads.

Table 1: Evaluation hardware and software setup.

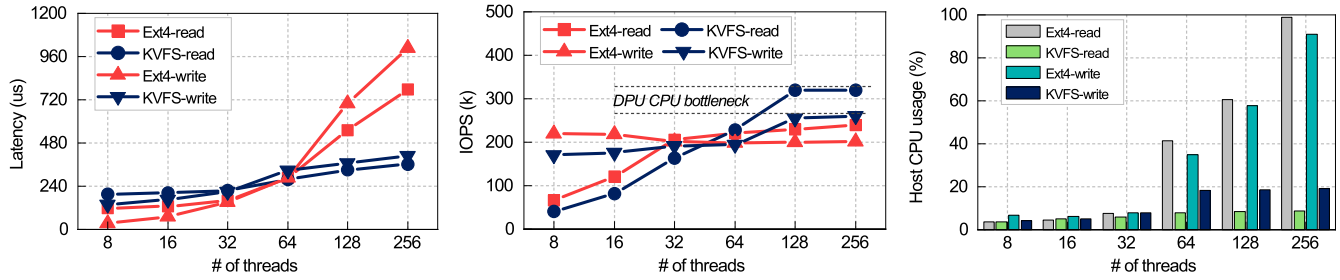
Component	Description
CPU	Intel Xeon Gold 6230R processors, 26 physical cores, 52 threads
Memory	128 GB DDR4
DPU	Huawei QingTian DPU card with 24 TaiShan cores at 2.0 GHz, 32 GB DRAM (max bandwidth: 32 GB/s), PCI 3.0 x16
OS and SW	Ubuntu 22.04 with kernel 6.2, vdbench 3.28, fio 3.36
NVMe SSD	Huawei ES3600P V5, read/write latency: 88μsec/14μsec

4.1 Raw Transmission Performance

To show the effectiveness of the proposed `nvme-fs` in DPC, we test the raw IOPS and data transmission latency. We use 4KB and 8KB as the data sizes for IOPS and latency evaluation, as most storage systems use 4K or 8KB as their smallest data granularity. To test the raw transmission performance, we implement a virtual client in DPU that responds to the requests from I/O dispatch with in-memory data. Thus, the transmission latency of `nvme-fs` in DPC and `virtio-fs` in DPFS can be regarded as the round-trip latency of host-DPU interaction.

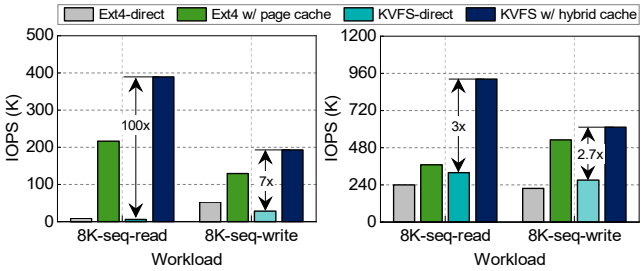
Figure 6 shows the data transmission IOPS and latency between `virtio-fs` and `nvme-fs` under different concurrency. The best read/write latency for `nvme-fs` is 20.6μsec/26.6μsec, while the number for `virtio-fs` is 36.5μsec/34μsec. Both of them achieve ultra-low latency when there is only one thread. As shown, `nvme-fs` consistently exhibits slightly lower latency than `virtio-fs` under low concurrency (# of threads <= 4). However, due to the inefficiency of the `virtio-fs` data path, which involves 2–3 times more DMA operations when compared to `nvme-fs`, `nvme-fs` outperforms `virtio-fs` by around 2 - 3 times in terms of both IOPS and latency when the number of threads increases. The peak performance of both `nvme-fs` and `virtio-fs` comes at 32 concurrent threads. We argue this is because concurrent threads that exceed the number of physical cores (i.e., 24 cores for our DPU) bring extra scheduling overheads that would hurt the performance.

We also evaluate the bandwidth of `virtio-fs` and `nvme-fs` with 1MB sequential read and write under 16 concurrent threads. `virtio-fs` achieves 5.1GB/s for writing and 6.3GB/s for reading. Since the current implementation of `virtio-fs` does not support multi-queue, limiting the bandwidth achieved in the test. On the



(a) Latency (b) IOPS (c) Host CPU usage

Figure 7: The latency, IOPS, and CPU usage comparison between local Ext4 and KVFS. ***We use 8K random read and write with direct I/O for big files in the test.***



(a) Single thread (b) 32 threads

Figure 8: Contributions of hybrid cache to random IOPS.

contrary, `nvme-fs` almost saturate the bandwidth of the PCIe interface (i.e., PCIe3.0 x16, around 15.7GB/s), with read and write bandwidths of 15.1GB/s and 14.3GB/s, respectively.

4.2 Standalone File Performance

Figure 7 compares the latency, IOPS, and CPU usage of 8K file random read/write operations between local Ext4 and KVFS. Note that we use only one local NVMe SSD for the performance test of local Ext4. Due to the extra overheads for the host-DPU interaction, KVFS exhibits lower IOPS and higher latency than local Ext4 in low to medium concurrency (i.e., # of threads <= 32). When the number of current threads exceeds 64, KVFS outperforms local Ext4 in both latency and IOPS, as shown in Figure 7(a) and Figure 7(b). Due to disk I/O contention and scheduling, the read and write latency for local Ext4 with 256 concurrent threads reaches 779μsec and 1009μsec, respectively. Besides, when the number of concurrent threads is higher than 32, the IOPS of local Ext4 reaches the limit of NVMe SSD and does not increase again. In contrast, KVFS converts file operations to remote KV operations. Its performance can easily scale with high-performance KV stores. The read and write latency for KVFS are 363μsec and 410μsec, respectively, which are still in an acceptable range. The IOPS of KVFS scales with the concurrency until 128 threads, where we find that the CPU usage of DPU reaches 100% and the IOPS is bottlenecked by the DPU's CPU. Therefore, we believe higher performance DPU can bring more file performance.

Table 2: Bandwidth comparison.

	Workloads	Ext4	KVFS
Single thread	1MB seq. read	1.8GB/s	5.0GB/s
	1MB seq. write	1.6GB/s	3.1GB/s
32 threads	1MB seq. read	3.0GB/s	7.6GB/s
	1MB seq. write	2.0GB/s	5.0GB/s

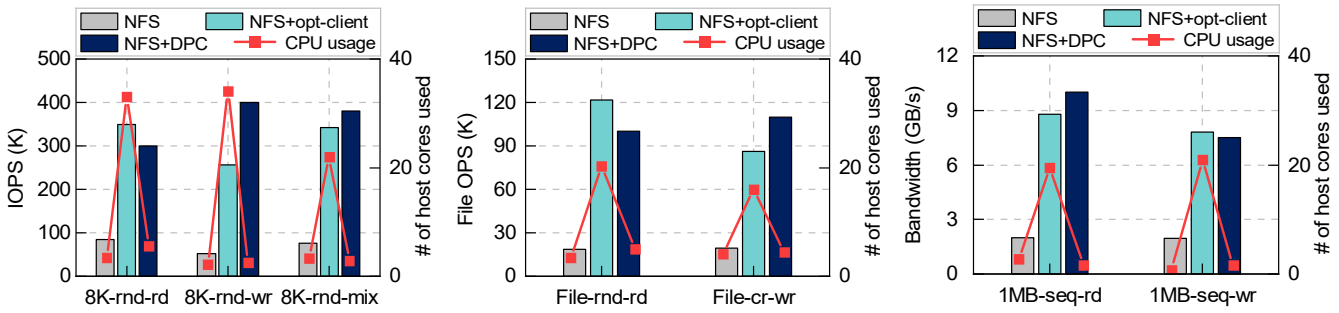
Figure 7(c) compares the host CPU usage between local Ext4 and KVFS. Since KVFS offloads cache management and file operations into DPU, KVFS exhibits much lower host CPU usage than local Ext4. As shown, the host CPU usage of KVFS is lower than 20% for all the concurrency degrees. In contrast, local Ext4 consumes a huge amount of host CPU cycles when the number of threads exceeds 32. With 256 threads, local Ext4 exhibits more than 90% of host CPU usage. On average, KVFS respectively saves 86% and 65% CPU cycles for small read and write I/O when compared to local Ext4 under high concurrency (i.e., # of threads >= 64).

To show the effectiveness of the proposed hybrid cache, we test the random IOPS in direct and buffered scenarios. As shown in Figure 8, both Ext4 and KVFS can benefit from the local cache. For Ext4, we use the default page cache to accelerate small I/Os. For KVFS, in addition to caching small writes, we actively prefetch data for sequential reads, boosting read IOPS by 100× with a single thread and 3× with 32 threads. By offloading the cache control plane into DPU, we can utilize customized cache replacement and prefetching algorithms tailored to the workload characteristics to improve I/O performance.

Besides read/write IOPS, we also evaluate the bandwidth of KVFS. Table 2 compares the sequential read and write bandwidth between local Ext4 and KVFS. KVFS outperforms Ext4 in both single-thread and multi-thread scenarios. We find that the read/write bandwidth is limited by the read/write performance of our disaggregated KV store. In our design, the standalone file read/write is shifted to the disaggregated storage. Thus, we believe that a high-performance disaggregated storage system can further improve the bandwidth of local standalone file operations.

4.3 Shared Distributed File Performance

To make distributed file systems benefit from the proposed DPC, we move EC calculation, I/O forwarding, cache management, DIO, and file delegation from host-side fs-client to DPC. To show the effectiveness of the proposed DPC, we evaluate the distributed file system performance with three types of fs-client: 1) standard NFS client (`NSF`); 2) NFS-compatible optimized host-side fs-client (`NSF+opt-client`); 3) NFS-compatible DPC (`NSF+DPC`). In the evaluation, we design three test cases: 1) 8K random read/write IOPS on big files (i.e., file size larger than 1GB), 2) small file operations performance, including 8K file random read and 8K file creation write; 3) file sequential read/write bandwidth.



(a) 8K IOPS on big files (b) 8K file operations (c) File bandwidth

Figure 9: Contributions of DPC to file performance and host-side CPU usage reduction. ***Rnd: random; seq: sequential; rd: read; wr: write; cr: create.***

Figure 9 shows the evaluation results. The standard NFS exhibits very low CPU usage (i.e., 1–3 cores) but delivers the lowest performance. The optimized fs-client achieves 4–5× IOPS improvements over the standard NFS client while also consuming 6–15× more CPU cores. Similarly, for file operation performance and bandwidth, the optimized fs-client also shows significant improvements over the standard NFS client but brings 11× more CPU usage on average.

As shown in Figure 9, the proposed DPC achieves similar CPU usage to the standard NFS client and comparable performance to the optimized fs-client. In some cases, such as 8K random write and 8K file creation write, DPC outperforms the optimized fs-client by around 40%. Compared to the optimized fs-client, DPC reduces CPU usage by around 90%. For example, the optimized fs-client typically consumes 30 CPU cores during the IOPS test, whereas the DPC only uses an average of 3.6 CPU cores. Compared to the standard NFS, DPC exhibits only about a 10% increase in CPU usage but delivers over 5× performance improvements.

5 RELATED WORK

File system client-side optimization. Client-side optimizations for file systems, especially distributed file systems, show high potential for performance improvement. Specifically, dedicated fs-clients become an essential part of distributed file systems and play an important role in achieving high performance. LustreFS [11, 29] enables data compression and write-back cache in fs-client to reduce the network traffic. BatchFS [43], IndexFS [32], LocoFS [23], and NFS v4 [27] employ the delegation mechanism to cache metadata and permissions on the client to reduce metadata server accesses. MetaWBC [28] proposes a metadata write-back caching mechanism to improve the performance of metadata operations in distributed file systems. Qian et al. propose `autoIO` [30], which automatically decides whether file I/O requests should be handled as buffered or direct I/O in Lustre client. Orion [42] and Assise [4] build a client-local file system. Orion manages file data in local PM (persistent memory) to reduce remote access. Assise places both file data and metadata in the client node's PM to eliminate the need for accessing remote metadata servers. Our work proposes a generic approach to make these optimizations be offloaded to DPU to ease the host-side CPU loads.

DPU offloading. Both SmartNIC and DPU play an important role in distributed systems. Many researchers have explored offloading host tasks to SmartNIC and DPU. hXDP [7] and OXDP [37] offload XDP (eXpress Data Path) to SmartNICs for accelerating packet processing and freeing the host-side CPU cycles. Since using RDMA is complicated and only provides a low-level abstraction, DFI [35] and DPI [2] propose the flow-based data processing interface to make distributed systems easily exploit the emerging capabilities of SmartNICs. To facilitate data storage, SKV [34] offloads data replication to the SmartNICs by separating background processing from client interaction on the server. DComp [12] offloads LSM-tree compaction to DPU to alleviate the CPU overhead. Girolamo et al. propose to offload storage policies, such as authentication, replication, and erasure coding, to SmartNICs [10]. The most related works to our paper are 1) DPFS, 2) LineFS, and 3) Fisc. DPFS [15] offloads the `virtio-fs` backends to DPU for virtualization environments using the `virtio-fs` protocol; LineFS [20] proposes a parallel datapath execution pipeline for offloading DFS operations to SmartNICs; Fisc [22] offloads security-related and network-related functions to DPU for the cloud-native file system to reduce the CPU and memory usage of computation servers. Following this direction, our work focuses on fs-client offloading and differs from these works in three aspects: 1) We propose a `nvme-fs` protocol for file operations offloading; 2) DPC supports both distributed and local standalone file systems; 3) DPC features the hybrid cache that can benefit from the near-application host-side caching.

6 CONCLUSION

In this paper, to free up the host CPU cycles consumed by high-performance fs-client, we propose DPC, a DPU-accelerated high-performance file system client. First, to effectively offload file operations to DPU, we propose `nvme-fs`, a high-performance NVMe-based file protocol for the DPU-offloaded file systems. Second, to accelerate file access, we propose a hybrid cache that offloads the cache control plane to the DPU while leaving the cache data plane to reside on the host side. Finally, to eliminate the low disk utilization issue in cloud environments, we propose KVFS, which allows DPC to provide standalone file services to replace local file systems like Ext4. We evaluate the proposed DPC with various workloads, experimental results show that DPC can outperform the state-of-art DPFS by 2–3 times in terms of both IOPS and latency. The proposed KVFS in DPC can also outperform Ext4 and reduce over 80% CPU usage in high concurrency scenarios. Besides, DPC-enabled DFS can achieve 90% host-side CPU usage reduction while maintaining high performance.

ACKNOWLEDGMENTS

The work described in this paper is partially supported by the Fundamental Research Funds for the Central Universities (2023CDJKYJH026, 2024CDJGF-003 and 2024CDJGF-032).

REFERENCES

[1] Alibaba. 2023. The High Performance Distributed File System by Alibaba Cloud. https://www.alibabacloud.com/blog/.
[2] Gustavo Alonso, Carsten Binnig, Ippokratis Pandis, Kenneth Salem, Jan Skrzypczak, Ryan Stutsman, Lasse Thostrup, Tianzheng Wang, Zeke Wang, and Tobias Ziegler. 2019. DPI: the data processing interface for modern networks. *CIDR 2019 Online Proceedings* (2019).
[3] Amazon. 2023. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.
[4] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. 2020. Assise: Performance and availability via client-local NVM in a distributed file system. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. 1011–1027.
[5] Jens Axboe. 2017. https://fio.readthedocs.io/en/latest/fio_doc.html.
[6] Sebastian Böhm and Guido Wirtz. 2021. Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes.. In *Proceedings of the 13th Central European Workshop on Services and their Composition (ZEUS'21)*. 65–73.
[7] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 973–990.
[8] Intel Corporation. 2023. DAOS. https://docs.daos.io/.
[9] NVIDIA Corporation. 2023. NVIDIA BlueField Networking Platform. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.
[10] Salvatore Di Girolamo, Daniele De Sensi, Konstantin Taranov, Milos Malesevic, Maciej Besta, Timo Schneider, Severin Kistler, and Torsten Hoefler. 2022. Building Blocks for Network-Accelerated Distributed File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. 1–14.
[11] Andreas Dilger. 2023. Lustre 2.16 and Beyond. https://wiki.lustre.org/images/d/d9/LUG2023-Lustre_2.16_and_Beyond-Dilger.pdf.
[12] Chen Ding, Jian Zhou, Jiguang Wan, Yiqin Xiong, Sicen Li, Shuning Chen, Hanyang Liu, Liu Tang, Ling Zhan, Kai Lu, and Peng Xu. 2023. DComp: Efficient Offload of LSM-tree Compaction with Data Processing Units. In *Proceedings of the 52nd International Conference on Parallel Processing (ICPP'23)*. 233—-243.
[13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. 51—-64.
[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 29–43.
[15] Peter-Jan Gootzen, Jonas Pfefferle, Radu Stoica, and Animesh Trivedi. 2023. DPFS: DPU-Powered File System Virtualization. In *Proceedings of the 16th ACM International Conference on Systems and Storage (SYSTOR'23)*. 1–7.
[16] Apache Hadoop. 2022. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[17] Huawei. 2023. OceanStor Scale-out Storage. https://e.huawei.com/en/products/storage/scale-out-storage.
[18] Fungible Inc. 2023. The Fungible DPU. https://www.hc32.hotchips.org/assets/program/conference/day2/HotChips2020_Networking_Fungible_v04.pdf.
[19] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 158–169.
[20] 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*. 756–771.
[21] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. 2017. UNO: Uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. 506–519.
[22] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, Zhaosheng Zhu, Huayong Wang, Haonan Qiu, Derui Liu, Shaozong Liu, Yujie Zhou, Yaohui Wu, Zhiwu Wu, Shang Gao, Chao Han, Zicheng Luo, Yuchao Shao, Gexiao Tian, Zhongjie Wu, Zheng Cao, Jinbo Wu, Jiwu Shu, Jie Wu, and Jiesheng Wu. 2023. Fisc: A Large-scale Cloud-native-oriented File System. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*. 231–246.
[23] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. 1–12.
[24] Marvell. 2023. Marvell LiquidIO III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.
[25] OpenSFS and EOFS. 2023. Lustre File System. https://www.lustre.org/.
[26] Oracle. 2023. https://wiki.lustre.org/VDBench.
[27] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. 2000. The NFS version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*.
[28] Yingjin Qian, Wen Cheng, Lingfang Zeng, Marc-André Vef, Oleg Drokin, Andreas Dilger, Shuichi Ihara, Wusheng Zhang, Yang Wang, and André Brinkmann. 2022. MetaWBC: Posix-compliant metadata write-back caching for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22)*. 1–20.
[29] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, et al. 2019. LPCC: hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. 1–14.
[30] Yingjin Qian, Marc-André Vef, Patrick Farrell, Andreas Dilger, Xi Li, Shuichi Ihara, Yinjin Fu, Wei Xue, and André Brinkmann. 2024. Combining Buffered I/O and Direct I/O in Distributed File Systems. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST'24)*. 17–33.
[31] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated smartnic offloading insights for network functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 772–787.
[32] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 237–248.
[33] Parminder Singh, Gurjot Balraj Singh, and Kiran Jyoti. 2015. A study on resource provisioning of multi-tier web applications in cloud computing. In *Proceedings of 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom'15)*. 799–802.
[34] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. 2022. SKV: A SmartNIC-Offloaded Distributed Key-Value Store. In *2022 IEEE International Conference on Cluster Computing (CLUSTER'22)*. 1–11.
[35] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The data flow interface for high-speed networks. In *Proceedings of the 2021 International Conference on Management of Data*. 1825–1837.
[36] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 59–72.
[37] Feng Wang, Gongming Zhao, Qianyu Zhang, Hongli Xu, Wei Yue, and Liguang Xie. 2023. OXDP: Offloading XDP to SmartNIC for Accelerating Packet Processing. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS'22)*. 754–761.
[38] 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)*. 987–1004.
[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 307—-320.
[40] WeKaIO. 2023. WekaFS Architecture. https://www.weka.io/wp-content/uploads/files/2020/08/weka-architecture-white-paper.pdf.
[41] Yan Yan, Arash Farhadi Beldachi, Reza Nejabati, and Dimitra Simeonidou. 2020. P4-enabled smart nic: Enabling sliceable and service-driven optical data centres. *Journal of Lightwave Technology* 38, 9 (2020), 2688–2694.
[42] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2019. Orion: A distributed file system for Non-Volatile main memory and RDMA-Capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. 221–234.
[43] Qing Zheng, Kai Ren, and Garth Gibson. 2014. BatchFS: Scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 2014 9th Parallel Data Storage Workshop (PDSW'14)*. 1–6.