DPFS: DPU-Powered File System Virtualization

Peter-Jan Gootzen*
IBM Research
Zurich, Switzerland
peter.jan.gootzen@ibm.com

Jonas Pfefferle
IBM Research
Yorktown Heights, USA
jpf@ibm.com

Radu Stoica
IBM Research
Zurich, Switzerland
rst@zurich.ibm.com

Animesh Trivedi
VU Amsterdam
Amsterdam, Netherlands
a.trivedi@vu.nl

ABSTRACT

As we move towards hyper-converged cloud solutions, the efficiency and overheads of distributed file systems at the cloud tenant side (i.e., client) become of paramount importance. Often, the client-side driver of a cloud file system is complex and CPU intensive, deeply coupled with the backend implementation, and requires optimizing multiple intrusive knobs. In this work, we propose to decouple the file system client from its backend implementation by virtualizing it with an off-the-shelf DPU using the Linux `virtio-fs` software stack. The decoupling allows us to offload the file system client execution to a DPU, which is managed and optimized by the cloud provider, while freeing the host CPU cycles. DPFS, our proposed framework, is 4.4× more host CPU efficient per I/O, delivers comparable performance to a tenant with zero-configuration and without modification of their host software stack, while allowing workload and hardware specific backend optimizations. The DPFS framework and its artifacts are publically available at https://github.com/IBM/DPFS.

CCS CONCEPTS

• **Networks** → Network File System (NFS) protocol; • **Software and its engineering** → **File systems management**; • **Information systems** → *Cloud based storage*; • **Hardware** → *Networking hardware*.

KEYWORDS

DPU, SmartNIC, Offloading, File system, Virtualization, Cloud, Storage, Framework, Datacenter, RDMA, NFS, Virtio-fs, FUSE

1 INTRODUCTION

File systems are a popular choice for cloud data storage with offerings such as traditional distributed file systems (HadoopFS, Ceph, GlusterFS), and *cloud-native file systems* (CNFS) services like Amazon EFS [3], AliBaba Pangu [10] or Azure Files [27]. With the recent push for hyper-converged infrastructure [13], there is a need for an efficient, scalable and high-performance cloud-native file system service.

Building a high-performance, scalable cloud-native file system for applications is a challenging task. First, the raw performance of storage and networking devices are constantly increasing, while the CPU performance improvements have stalled [29, 44]. As a result, delivering the full speed of I/O devices in a disaggregated storage setting takes a considerable amount of CPU resources [17, 43]. For example, Alibaba reports 12 CPU cores are required to deliver 200 Gbps of block-level traffic [26]. At the file system level, LineFS reports that with Ceph a single fully-utilized CPU only delivers ~10 Gbps bandwidth on a 100 Gbps link [16]. The question of CPU efficiency is also important for bare-metal machines, which have become popular in clouds recently [6, 34, 48]. Second, client-side CNFS logic can be complex and bloated, as it has to implement logic for communication and coordination with metadata and data servers, client-side buffer and connection management, caches, etc. As a result, it is not uncommon for distributed file system clients to consume GBs of DRAM and a significant amount of CPU cycles, thus limiting how many concurrent tenants (VMs, containers) can be packed on a server [2, 21]. Lastly, the close coupling of the file system API and its implementation makes it difficult to deploy new extensions or optimizations. For example, a bare-metal tenant using Ceph can not easily switch to HopsFS [30] or InfiniFS [25] without significant disruptions if it experiences metadata scalability challenges. Furthermore, many of these CNFS come with hundreds of performance knobs and features, which requires explicit deployment and optimizations from the tenant side to extract the best possible performance.

	ext4	ext4 + NVMe-oF	XFS	Btrfs
I/O operations	5.2	13.7	3	4.6
Total Bytes (in KiB)	44.7	46.8	12	125.3
Amplification	11.2x	11.7x	3x	16x

Table 1: Analysis of storage (block) or network (packets with NVMoF) operations for a single 4KiB file write.

To address the aforementioned challenges, we propose to *virtualize the access to a file system by offloading the file system client to a DPU* to offer a tenant-transparent, light-weight, high-performance file system service. Such a design has multiple advantages: First, virtualization decouples the file system API from its backend implementation, which enables us to optimize the backends to support multiple workload needs such as multiple APIs [22], scalable metadata lookups with KV stores, decoupling of data from metadata management [19]. A limited form of such decoupling is currently offered by cloud providers in the form of an NFS gateway to the CNFS client [3, 12, 27]. We argue this approach gives away control of the file system client from the cloud provider, and demonstrate that the Linux kernel NFS client has high overheads (§3.4). Second, by offloading (and leveraging the hardware acceleration of the DPU) the file system implementation, we free host CPU resources for the tenant. One can argue that offloading capabilities can also be leveraged by the host either at the block, or application level. A block-level offloading allows a fully offloadable I/O stack [20, 28],

*Also with VU Amsterdam.

SYSTOR '23, June 5–7, 2023, Haifa, Israel

ACM ISBN 978-1-4503-9962-3/23/06.
https://doi.org/10.1145/3579370.3594769

however, it suffers from significant I/O amplifications. We quantify this amplification [24] in Table 1, where we report the number of block-level or network-level operations (packets for NVMe-oF storage disaggregation) generated in response to a single 4 KiB file write operation. Depending on the file system, the amplification can be as high as 3-16×, thus requiring proportional gains from hardware offloading. Furthermore, in comparison to local block I/O, NVMe-oF also amplifies the average number of I/O operations needed (5.2 vs. 13.7). On the other hand, with file system-level virtualization, this will be a single offloadable file operation from the host to the DPU and then from the DPU to the file system backend. An application-level solution requires rewriting the application to benefit from offloading capabilities of the DPU, which are non-trivial and non-standardized (except for RDMA and NVMe-oF style offloading). Lastly, a DPU-powered design does not require any fine-tuning or configuration from the tenant side. All CNFS-related configuration and optimizations can be consolidated on the DPU, under the cloud provider's purview, freeing the host CPU for tenants. The physical separation of the DPU (which runs the cloud provider logic) from the tenant (on the host) also discourages any resource-sharing based side-channel attacks [6], thereby improving tenant isolation and security.

In this paper, we present DPFS, a DPU-Powered File System virtualization framework for cloud environments. DPFS leverages the `virtio-fs` protocol to communicate between the host and the DPU. The DPU and host communicate via PCIe memory-mapped `virtio` [38] queues using the FUSE file I/O command set. As `virtio-fs` is standardized [46] and included in the Linux kernel [37] (no installation required), this design enables DPFS to avoid the need for running a custom CNFS client on the host. The *de-coupling* of the front-end (a standard file system API) from its backend implementation (i.e., the CNFS) with a bump-in-the-write architecture on the DPU [9] allows the cloud provider to do a variety of network and storage optimizations (scheduling, caching, quotas, QoS), without having the tenant perform any code installation or change any configuration. To prove DPFS's flexibility, we have implemented three file system backends: (i) **`DPFS-null`**, a no-op DPU client useful for development and benchmarking; (ii) **`DPFS-NFS`**, a NFS backend that runs on the DPU and translates the `virtio-fs` requests into equivalent NFS commands and communicates with a remote NFS server using `libnfs` with Nvidia XLIO (partial TCP offloading) [33]; (iii) **`DPFS-KV`**, a RAMCloud-based backend that implements low-latency I/O operations on small files, a suitable fit for KV stores [35].

Our primary contributions in this work include:

- We make a case for transparent file system access virtualization for cloud-native file systems using DPUs.
- We present the design and implementation of DPFS, an open-source file system virtualization framework with support for the Nvidia BlueField-2 DPU [32] (https://github.com/IBM/DPFS).
- Evaluation of DPFS that demonstrates that it (i) is light-weight; (ii) delivers equal and better performance than a host NFS client; and (iii) is customizable and modular with multiple file system backend implementations.

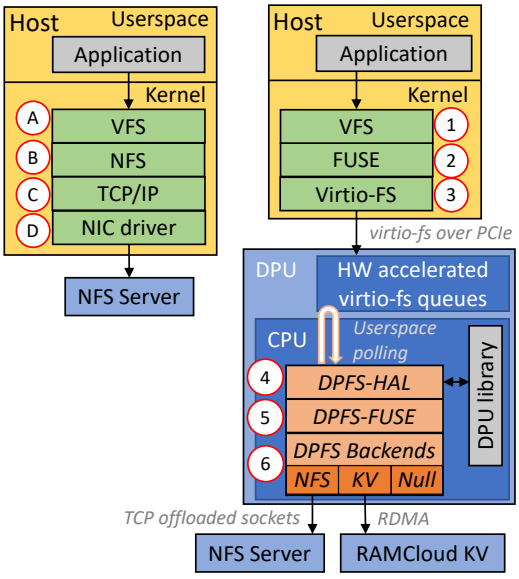


Figure 1: The architecture of DPFS compared to a host NFS client. Green boxes are the standard in-kernel code, and orange boxes are our contributions.

2 DPFS DESIGN AND IMPLEMENTATION

To accelerate the development of DPU-powered file systems, we implemented the DPFS framework. The framework is composed of three layers, each serving a different purpose, as shown in orange in Figure 1. The first layer, (`DPFS-HAL`) provides a Hardware Abstraction Layer (HAL) for the vendor-specific DPU functionality and the host-DPU optimizations. Its role is to reduce the DPU-specific knowledge required to develop a file system backend, to simplify code maintenance, and to allow a faster transitioning to a new type or new model of a DPU. The second layer implements a FUSE-like API (i.e., *libfuse* [1]) on top of the raw `virtio-fs` protocol buffers (`DPFS-FUSE` §2.2). The third layer contains the file system backend implementation (i.e., the DPU logic that allows connecting to the remote file system). Here we provide three implementations. The first client is `DPFS-null`, which implements a no-op backend that immediately replies to any request. This back-end is used for optimizing and benchmarking of the host-DPU communication, i.e., the `DPFS-HAL` and `DPFS-FUSE` layers. The second client is `DPFS-NFS` (§2.3), an optimized userspace NFS client that allows connecting to existing cloud NFS-based file system services. Using this client, a cloud provider would not have to perform any changes to existing file system back-ends to employ DPU virtualization. The third client is a key-value client (`DPFS-KV` §2.4) that connects to a RAMCloud [35] back-end and leverages RDMA to access files stored in the KV store. Past research has demonstrated that KV stores can be a better fit for file system implementations [18].

2.1 The DPFS I/O Path

In the host NFS client configuration (see Figure 1, left side), the application submits a file-related syscall (e.g., a *read()* call), that context switches to the kernel. The request traverses the VFS (A) and NFS (B) layers, then the TCP/IP stack (C), and is finally sent

to the NIC driver (D). The NIC communicates with the file system backend. Upon receiving a response, the reversed path is initiated by an interrupt.

On the other hand, the DPFS I/O path (see Figure 1, right side) is much more lightweight on the host, without requiring application or kernel modifications. The request first passes through the VFS layer (1) and is then forwarded to the lightweight FUSE layer (2) that transforms the VFS operation into a FUSE request message. The FUSE request is then passed to the `virtio-fs` layer (3). It encapsulates and transports the message over PCIe, through virtual functions to the appropriate host tenant and `virtio` queue on the DPU. On the DPU, a `DPFS-HAL` thread retrieves the message from the queue (4), then passes it to `DPFS-FUSE` (5). `DPFS-FUSE` extracts the original FUSE request and forwards it to the file system backend (6).

The complexity of the code path can be roughly quantified through lines of code in the Linux kernel (v6.2). Steps (B)-(C) contain 181k lines of source code (assuming IPv4), while steps (2)-(3) contain 13k lines of source code. On the DPU, the clients operate in userspace, thus no context switching is required. Furthermore, file system backends on the DPU can use hardware acceleration such as RDMA and TCP offload. Overall, the code path and the CPU overhead per operation on the host are significantly reduced, as we show in §3.4.

We emphasize that `virtio-fs` uses FUSE in such a manner that its traditionally large overheads are not incurred. The `virtio-fs` protocol uses FUSE for its standardized encoding of Linux Virtual File System (VFS) operations into the FUSE-ABI. Traditional FUSE encodes VFS operations and sends them back to userspace to be handled by a userspace file system implementation, incurring two extra context switches and significant overhead [47]. Where-as `virtio-fs` encodes the VFS operations using FUSE and sends them to the `virtio-fs` PCIe device (i.e. the DPU) directly from kernel space, thus incurring no extra context switches.

2.2 `DPFS-HAL` & `DPFS-FUSE` Implementation

One of the two main challenges when implementing a file system on top of the `virtio-fs` functionality of the DPU is to transparently handle the DPU's software stack. This includes configuring the software stack, the `virtio-fs` queues, implementing efficient polling and scheduling. For our DPU (the Nvidia BlueField-2), the firmware exposes the `virtio-fs` device in userspace as a RDMA device through *InfiniBand verbs*. Nvidia provides a library (Nvidia SNAP) to setup and configure `virtio` devices and to expose the `virtio-fs` functionality. However, `DPFS-HAL` (Hardware Abstraction Layer) hides such complexities by exposing only the `virtio-fs` relevant configuration parameters of the DPU's software stack. `DPFS-HAL`'s C-API allows the `virtio-fs` implementation to register a *handle_request* callback, poll on the `virtio` queues (in a thread managed by `DPFS-HAL` or manually) and complete `virtio-fs` requests in an asynchronous fashion.

Developing a file system backend using the `virtio-fs` protocol that `DPFS-HAL` exposes is cumbersome, as the file system developer must consume the raw FUSE-ABI that the Linux kernel exposes over the `virtio-fs` protocol. The FUSE-ABI has limited documentation as it is meant to be consumed through the popular *libfuse* library [1], which acts as the reference implementation and API for the ABI. `DPFS-FUSE` exposes a C-API that is similar to that of the *libfuse* API. By exposing a well-known and commonly used API, we report increased `virtio-fs` file system development speed. A file system developer could use userspace FUSE as a first step during development and transition to a DPU environment when available. Our `DPFS-FUSE` API builds on top of *libfuse* with added support for asynchronous operations and optimized function calls to decrease the number of data copies and allocations.

CPU	2x Intel Xeon E5-2630 v3, 2.4GHz, 8 cores/socket, C-states disabled during latency experiments
DRAM	128 GiB, DDR4 1,866 MT/s
DPU and network	Nvidia BlueField-2 with 2.75 GHz x 8 ARMv8 A72 64-bit cores, 16 GiB DRAM (max bandwidth: 18 GiB/s), connected via PCI 3.0, 100 Gbps link to a remote server, running BlueField DPU OS 3.9.3 with a prototype firmware to enable `virtio-fs` support
Host and remote server SW	Ubuntu 22.04 with kernel 6.2, fio 3.28, NFS (v4.2, NFS server params "async, no_subtree_check", NFS client params: "async")

Table 2: Benchmarking hardware and software setup.

2.3 `DPFS-NFS`: Hardware-accelerated NFS Backend

The `DPFS-NFS` backend implementation leverages the accelerated socket API of Nvidia XLIO [33] and the *libnfs* library [39]. Nvidia XLIO is a shared library that overloads the socket C-API to leverage the network offloading capabilities in the BlueField-2 hardware with reduced data copies (not fully zero-copy). This implementation showcases the possibility to leverage hardware acceleration through the DPU without any changes to the host file system client and the remote file system provider. Our experiments show the performance benefits of implementing an userspace hardware-accelerated NFS client. The standard Linux kernel NFS client (running on the DPU), when using *io_uring*, incurs 100.5 μsec read and 101.9 μsec write latencies for 4 KiB accesses. The software-only *libnfs* attains 76.0 μsec and 74.2 μsec respectively. Adding Nvidia XLIO to *libnfs* further reduces latency to 52.9 μsec and 52.2 μsec, respectively.

On the `DPFS-HAL` queue poller thread, the `DPFS-NFS` file system implementation translates the FUSE request into NFS v4.1 [31] and sends the NFS request to the remote storage backend using *libnfs* and Nvidia XLIO. A second thread polls on the Nvidia XLIO socket using *libnfs*, translates NFS v4.1 responses back into FUSE, and messages the `DPFS-HAL` poller thread to complete the `virtio-fs` request.

2.4 `DPFS-KV`: RAMCloud KV store Backend

The file system transparency of DPFS allows one to consume a specialized file system without the need to make changes to the host. We demonstrate this by implementing a DPFS backend for RAMCloud, a distributed, in-memory KV store that provides low-latency access by leveraging RDMA [35]. `DPFS-KV` maps file system operations from `DPFS-FUSE` to KV operations by exposing a set of key-value pairs through the file system's root directory, where each file represents a key-value pair. Two tables are stored in RAMCloud, the first to store data (file contents) and the second to store metadata (attributes, name, etc.). To index into the metadata table, the file

name is hashed and used as the key. In the metadata table, a unique inode identifier is stored, which acts as the file's key for the data table. To support listing the contents of the flat file system directory (i.e. *readdir()*), the file name itself is stored in the metadata table.

3 EVALUATION

In this section, we evaluate the performance and efficiency of DPFS through the `DPFS-null`, `DPFS-NFS` and `DPFS-KV` backends. Table 2 shows our experimental setup. All workloads are generated using *fio* with the *io_uring* I/O engine [7]. To generate a consistent load on the system, we use two *fio* threads.

3.1 Virtualization Overheads

We first measure the throughput of the `DPFS-null` backend for random read and write workloads (10 seconds warm up, followed by a 60 seconds run). The results of this experiment demonstrate the upper bound on the performance DPFS can deliver.

Figure 2a shows throughput of the null-DPU backend (y-axis, in GiB/s) vs. the I/O (queue) depth. We report throughput for 4 KiB, 16 KiB, and 64 KiB block sizes for both read and write operations. Throughput scales for both read and write operations as we increase the I/O depth. It plateaus when using a 64 KiB block size at queue depth between 16-32 at 5.87 GiB/s for read and 4.32 GiB/s for write operations. For a 4 KiB block size, a common I/O size in the kernel, the peak read throughput is 1,278 MiB/s and write throughput is 938 MiB/s, equivalent to 327K and 240K random 4 KiB IOPS respectively. At an I/O depth of 1, we measure a read throughput of 193 MiB/s and a write throughput of 168 MiB/s, which translates to a 38.6 μs and 43.3 μs latency baseline DPFS overhead respectively.

3.2 `DPFS-NFS` Performance

Having established the virtualization overheads (via the peak performance), we now evaluate the complete I/O path to a remote NFS server from the host versus from the DPU (steps ①-⑥ vs. Ⓐ-Ⓓ in Figure 1) when using 4 KiB block sizes (10 seconds warm up, followed by a 60 seconds run). We select 4 KiB as it is one of the most common block sizes, and is the I/O request size that also matches the granularity of the system memory pages. 4 KiB I/O sizes further stresses the software overheads (memory allocation, scheduling costs), hence, highlighting any inefficiencies.

Figure 2b shows the throughput (y-axis in MiB/s) vs. the I/O depth (x-axis). A host NFS client (path Ⓐ-Ⓑ) delivers better (11%-21%) read and write throughput up to an I/O depth of 3. However, above an I/O depth of 4, DPFS benefits from having a partially offloaded networking stack, and delivers better throughput (12%-32%) than the host NFS. However, all configurations do not scale beyond I/O depth of 16 (see limitations §3.5).

We further analyze the latency of 4 KiB I/O operations in Figure 2c. As discussed above, at an I/O depth of 1, the host NFS is faster than `DPFS-NFS`. For `DPFS-NFS`, we break down the overhead into two parts, the DPU induced latency (shown at the bottom bars, 38.6-43.3 μsec) and the software latency (the upper part, 52.9-52.2 μsec). We expect the former latency to improve with the newer generation of DPUs. Furthermore, we use *libnfs* with Nvidia XLIO that emulates socket semantics. We expect, a native *libnfs* implementation using RDMA would be able to also lower the userspace NFS overheads. The userspace NFS latency (the latency of NFS from the DPU, the top parts) is 26.7%-33% lower than the host NFS latency due to the hardware offloaded network stack used by `DPFS-NFS`.

	NFS	DPFS-KV
Read	71.2μs (σ: 10μs)	62.6μs (σ: 19.4μs)
Write	79μs (σ: 12.7μs)	70.79μs (σ: 21.2μs)

Table 3: Host NFS vs. `DPFS-KV` 4 KiB file I/O latencies.

	NFS	DPFS-NFS	+/-
Instructions/op	88,453	32,907	-62.80%
IPC	0.57	0.94	+64.21%
Branch miss rate	2.02	1.06	-47.42%
L1 dCache miss rate	8.82	3.82	-56.65%
dTLB miss rate	0.14	0.15	+7.14%
Savings in CPU cycles/op		**4.4×**	

Table 4: The microarchitectural profile of the host CPU.

3.3 `DPFS-KV` Performance

Lastly, we demonstrate that by specializing the file storage backend with RAMCloud, DPFS can deliver performance optimizations to workload-specific deployments. Table 3 shows our results of accessing complete 4 KiB files which are stored in NFS and RAMCloud (5 seconds warm up, followed by a 10 second run). These files are accessed in their entirety (i.e. a single I/O operation). In comparison to host NFS, `DPFS-KV` offers 7-12% latency gains over read and write operations.

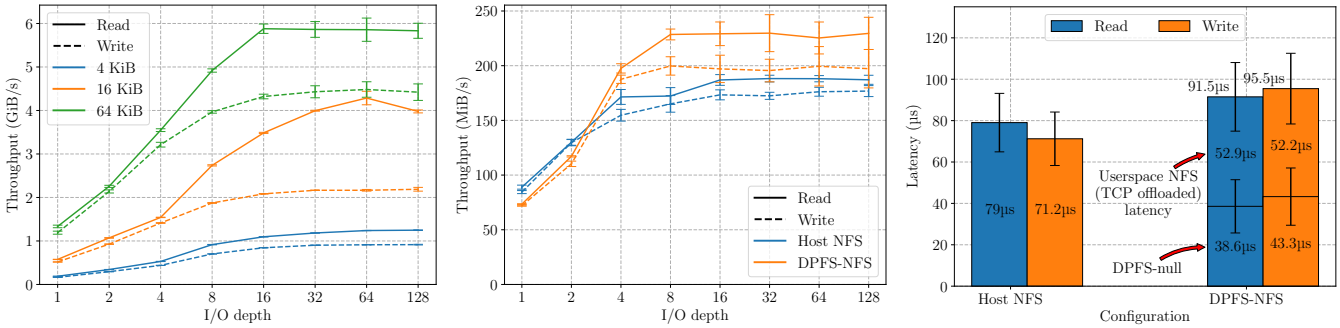
3.4 Host microarchitectural analysis

To explain the efficiency gains with DPFS, we conduct a low-level microarchitectural analysis. We measure the instructions/operations (quantifies the software overheads in the I/O path), IPC (quantifies the code path efficiency), branch missprediction rates (quantifies how predictable the code path is), L1 dCache miss rate (quantifies data fetch rates), and lastly, dTLB miss rates (quantifies memory management related overheads). All events are measured using *perf* with the `-kernel-only` flag on the *host CPU*, hence, comparing the cost of in-kernel host NFS with `DPFS-NFS`. We first measure an idle machine for 10 minutes (5 runs, averaged), and then with a random 4 KiB read/write (50/50 distribution) using an I/O depth of 128 workload for 10 minutes (5 runs, averaged). By subtracting the two, we quantify the cost of the host file system client implementations of host NFS and DPFS (i.e. `virtio-fs`, `DPFS-NFS` on the DPU).

Table 4 shows that (1) `DPFS-NFS` is light-weight compared to the host NFS, requiring 62.80% less instructions to complete a random 4 KiB operation; (2) it has better IPC, partially due to the smaller code path, but also due to the polling nature of the `DPFS-NFS` request dispatching and processing; (3) `DPFS-NFS` has lower branch and L1 dCache miss rate, implying a simplified execution profile. It has marginally higher dTLB miss rates, which we attribute to the inefficient page usage of `virtio-fs` (see §3.5). Overall, by combining the instructions per I/O and IPC, we find that *the DPFS host I/O path (steps 1-6) is* 4.4× *more efficient than host NFS (steps A-D).*

3.5 Limitations and Future Work

The current work is limited by a few restrictions, which we expect to be improved as the usage of `virtio-fs` becomes prevalent. Firstly,



(a) `DPFS-null` random I/O throughput scaling with I/O depth.

(b) Host NFS and `DPFS-NFS` random 4 KiB I/O throughput.

(c) Host NFS, `DPFS-NFS` and `DPFS-null` latency breakdown for random 4 KiB I/O

Figure 2: DPFS performance analysis

the current implementation of `virtio-fs` in the kernel does not support multi-queue [37], hence, DPFS currently only supports a single `virtio` queue per tenant. Because of an implementation limitation in DPFS we can only leverage a single DPU thread to process the single `virtio` queue, limiting the throughput achieved in Figure 2a and Figure 2b. A single queue only limits the available throughput per tenant, not the number of tenants supported. We are currently working on adding support for multi-tenancy through SR-IOV [8] on the Nvidia BlueField-2 to DPFS.

Secondly, the FUSE headers use an excessive amount of pages (3 pages for a read I/O, 4 pages for a write I/O) to store their headers [37]. Due to a memory bottleneck of Nvidia XLIO (tight integration with hardware, huge pages, memory pinning) when using large amounts of memory, `DPFS-NFS` can only configure the `virtio` queue size with a maximum of 64 pages in the DPU. With at least four pages overhead in a `virtio-fs` operation, `DPFS-NFS` is therefore restricted to a queue depth of only 16 (64/4).

Thirdly, since `virtio-fs` file system operations flow through the DPU's SoC complex, the comparably weak hardware found in the Nvidia BlueField-2 (ARM A72 cores, single-channel DDR3) [32] poses a significant bottleneck. This bottleneck can be seen in the throughput difference of `DPFS-null` (Figure 2a) and `DPFS-NFS` (Figure 2b). Where `DPFS-NFS` is only able to achieve ~20% of the 4 KiB throughput of `DPFS-null` because of TCP processing and the several extra data copies it incurs.

We expect that the next iterations of the hardware/software stack will help alleviate these limitations, and that tightly integrating the file system implementation with the hardware to prevent memory copies will speed up large block size workloads.

4 RELATED WORK

All cloud providers offer high-performance, scalable file-system services designed for large scale, multi-tenancy, and high availability. However, today, the typical way of accessing cloud file-system services is either through an NFS client (standard in-kernel) [3, 12, 27] or by running a cloud-provider's file system client code on the tenant machine [10, 11, 36]. Such an approach provides poor isolation between cloud tenants and providers, while consuming significant tenant resources (CPU, RAM). With improving DPU capabilities [4, 5], many projects in both academia and industry explored leveraging them for virtualizing and accelerating various cloud services. Examples include for networking functions [9, 41, 42, 49], for block storage [14, 15, 26, 28], for key-value and object storage [23, 40, 45]. Our work follows in this direction by focusing on offloading and abstracting file-system services, an area overlooked until recently.

The closest works to our paper are (1) LineFS [16], which provides a persistent memory distributed file system that leverages the DPU for offloading noisy background tasks, and (2) FISC [21], which is a cloud-native file system that offloads the complete file system to an FPGA-based DPU and remote servers through a custom Virtio-Fisc device interface. DPFS differs from these works in two manners. Firstly, DPFS and FISC utilize DPU-offloading for virtualization, by decoupling the file system client from the file system implementation, the burden on the host CPU is decreased and the control the Cloud provider has over the file system stack is increased. Whereas LineFS utilizes DPU-offloading for performance improvements. Secondly, DPFS differs from FISC in that it offloads the file system implementation to an off-the-shelf CPU-based DPU (e.g. Nvidia BlueField-2) with the existing `virtio-fs` host software-stack and provides an open API for implementing different file system backends.

5 CONCLUSIONS

To address the challenges of providing a light-weight client for distributed file systems, we propose to virtualize and decouple the file system API from its implementation using DPUs. We leverage the Linux `virtio-fs` software stack to pass file operations to a DPU, where various optimized network/storage backends can be implemented by a cloud provider. In our evaluation we demonstrate that DPFS, our framework, is light-weight (40 μsec base latency, 4.4× lower host CPU overhead), has competitive performance, and requires zero configuration or modifications on the host.

ACKNOWLEDGMENTS

We would like to thank our shepherd Youyou Lu and our anonymous reviewers for their valuable feedback. We would also like to thank Nvidia for providing us the BlueField-2 prototype firmware that provided `virtio-fs` support. Animesh Trivedi is supported by the Dutch Research Council (NWO) grant number OCENW.KLEIN.561.

REFERENCES

[1] 2023. Libfuse: The Reference Implementation of the Linux FUSE (Filesystem in Userspace) Interface. https://github.com/libfuse/libfuse

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[3] Amazon AWS. 2023. *What is Amazon Elastic File System? - Amazon Elastic File System.* https://docs.aws.amazon.com/efs/latest/ug/whatisefs.html

[4] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. 2021. Intel's Hyperscale-Ready Infrastructure Processing Unit (IPU). In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–16. https://doi.org/10.1109/HCS52781.2021.9567455

[5] Idan Burstein. 2021. Nvidia Data Center Processing Unit (DPU) Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–20. https://doi.org/10.1109/HCS52781.2021.9567066

[6] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 599–613. https://doi.org/10.1145/3037697.3037703

[7] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A Systematic Study of Libaio, SPDK, and Io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '22)*. Association for Computing Machinery, New York, NY, USA, 120–127. https://doi.org/10.1145/3534056.3534945

[8] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. 2010. High performance network virtualization with SR-IOV. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–10. https://doi.org/10.1109/HPCA.2010.5416637

[9] 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* (Renton, WA, USA) *(NSDI'18)*. USENIX Association, USA, 51–64.

[10] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[11] Google. 2021. *Colossus under the hood: a peek into Google's scalable storage system.* https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system

[12] Google Cloud. 2023. *Technical overview - Google Cloud Filestore.* https://cloud.google.com/filestore/docs/overview

[13] Karan Gupta. 2020. From Hyper Converged Infrastructure to Hybrid Cloud Infrastructure. USENIX Association.

[14] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 127–140. https://www.usenix.org/conference/nsdi20/presentation/hwang

[15] Junbin Kang, Chunming Hu, Tianyu Wo, Ye Zhai, Benlong Zhang, and Jinpeng Huai. 2016. MultiLanes: Providing Virtualized Storage for OS-Level Virtualization on Manycores. *ACM Trans. Storage* 12, 3, Article 12 (jun 2016), 31 pages. https://doi.org/10.1145/2801155

[16] 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* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 756–771. https://doi.org/10.1145/3477132.3483565

[17] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. *SIGARCH Comput. Archit. News* 45, 1 (apr 2017), 345–359. https://doi.org/10.1145/3093337.3037732

[18] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. 2021. Modernizing File System through In-Storage Indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 75–92. https://www.usenix.org/conference/osdi21/presentation/koo

[19] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. https://doi.org/10.1109/MSST.2015.7208288

[20] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 591–605. https://doi.org/10.1145/3373376.3378531

[21] 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* (Santa Clara, CA, USA) *(FAST'23)*. USENIX Association, USA, Article 15, 15 pages.

[22] Kunal Lillaney, Vasily Tarasov, David Pease, and Randal Burns. 2019. Agni: An Efficient Dual-Access File System over Object Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 390–402. https://doi.org/10.1145/3357223.3362703

[23] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 363–378. https://www.usenix.org/conference/atc19/presentation/liu-ming

[24] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the Lifetime of Flash-Based Storage through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'13)*. USENIX Association, USA, 257–270.

[25] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. 2022. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 313–328. https://www.usenix.org/conference/fast22/presentation/lv

[26] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. 2022. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 753–766. https://doi.org/10.1145/3544216.3544238

[27] Microsoft Azure. 2023. *Azure Files - Managed File Shares and Storage.* https://azure.microsoft.com/en-us/products/storage/files

[28] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 106–122. https://doi.org/10.1145/3452296.3472940

[29] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2015. Non-Volatile Storage: Implications of the Datacenter's Shifting Center. *Queue* 13, 9 (nov 2015), 33–56. https://doi.org/10.1145/2857274.2874238

[30] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 89–104. https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi

[31] David Noveck and Chuck Lever. 2020. Network File System (NFS) Version 4 Minor Version 1 Protocol. RFC 8881. https://doi.org/10.17487/RFC8881

[32] NVIDIA. 2021. NVIDIA BlueField-2 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf

[33] NVIDIA. 2023. NVIDIA Accelerated IO (XLIO) Documentation. https://docs.nvidia.com/networking/display/XLIOv214

[34] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. 2015. Improving Agility and Elasticity in Bare-Metal Clouds. *SIGARCH Comput. Archit. News* 43, 1 (mar 2015), 145–159. https://doi.org/10.1145/2786763.2694349

[35] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (aug 2015), 55 pages. https://doi.org/10.1145/2806887

[36] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[37] Red Hat, Inc. 2022. *virtio-fs Linux Kernel implementation - Linux kernel source tree v6.2.* https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/fuse/virtio_fs.c?h=v6.2

[38] Rusty Russell. 2008. Virtio: Towards a de-Facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (jul 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[39] Ronnie Sahlberg. 2023. Libnfs: NFS client library. https://github.com/sahlberg/libnfs

[40] Giuseppe Siracusano and Roberto Bifulco. 2017. Is It a SmartNIC or a Key-Value Store? Both!. In *Proceedings of the SIGCOMM Posters and Demos* (Los Angeles, CA, USA) *(SIGCOMM Posters and Demos '17)*. Association for Computing Machinery, New York, NY, USA, 138–140. https://doi.org/10.1145/3123878.3132014

[41] Brent Stephens, Aditya Akella, and Michael M. Swift. 2018. Your Programmable NIC Should Be a Programmable Switch. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 36–42. https://doi.org/10.1145/3286062.3286068

[42] Maroun Tork, Lina Maudlej, and Mark Silberstein. 2020. Lynx: A SmartNIC-Driven Accelerator-Centric Architecture for Network Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 117–131. https://doi.org/10.1145/3373376.3378528

[43] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Kornilios Kourtis, Ioannis Koltsidas, and Thomas R. Gross. 2018. FlashNet: Flash/Network Stack Co-Design. *ACM Trans. Storage* 14, 4, Article 30 (dec 2018), 29 pages. https://doi.org/10.1145/3239562

[44] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. 2018. Albis: High-Performance File Format for Big Data Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 615–630. https://www.usenix.org/conference/atc18/presentation/trivedi

[45] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 33–48. https://www.usenix.org/conference/atc20/presentation/tsai

[46] Michael S. Tsirkin and Cornelia Huck. 2022. Virtual I/O Device (VIRTIO) Version 1.2. https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html

[47] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 59–72. https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[48] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-Density Multi-Tenant Bare-Metal Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 483–495. https://doi.org/10.1145/3373376.3378507

[49] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 61, 18 pages.