Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

DPFS++: Cloudifying the DPU-Powered File System Virtualization Framework

---

Author: Peter-Jan Gootzen (2703924)

1st supervisor:	dr. ir. Animesh Trivedi	Vrije Universiteit Amsterdam
daily supervisor:	ir. Jonas Pfefferle	IBM Research, Yorktown Heights
daily supervisor:	dr. dipl. ing. Radu Stoica	IBM Research, Zurich
2nd reader:	dr. ir. Tiziano De Matteis	Vrije Universiteit Amsterdam

A thesis submitted in fulfillment of the requirements for
the joint UvA-VU Master of Science degree in Computer Science
Parallel Computing Systems track

September 29, 2023

"We can solve any problem by introducing an extra level of indirection."
David J. Wheeler

Abstract

Cloud data centers are the backbone of our modern-day digital society. These large buildings, full of computing power and networking capabilities, run workloads that support all aspects of our daily lives. As our demand for computing power has increased, data center operators have shifted towards heterogeneous architectures for computing that can provide more computing power for less energy.

Part of this heterogeneous evolution has been happening in the networking side of the server machine with so-called Data Processing Units (DPUs) (sometimes called SmartNICs). These pieces of hardware are network interfaces with offload capabilities baked in and allow data center operators to provide services to tenants in an isolated and performant manner. Today, all major cloud providers deploy DPUs for cloud services like block storage, networking, and data center orchestration.

Several academic research papers have proposed designs and prototypes for offloading a file system storage service to the DPU. In this thesis, we answer the following scientific question: Can DPFS provide a cloud-native solution for DPU-offloaded file system services? We analyze the file system service design space and identify currently untested designs. Through design, implementation, and evaluation, we also propose DPFS++, an augmented version of the established DPFS framework (51) for building DPU-powered file system services. DPFS++ has the following enhancements: (1) performance improvements, (2) a previously unexplored system design, (3) a backend for consuming existing cloud storage infrastructure (e.g., Ceph), and (4) provide dynamic multi-tenancy support.

With these extensions, we report that DPFS++ can satisfy the multi-tenancy requirements of a DPU-powered file system service in a cloud data center, and that its performance is on par with current NFS-based cloud solutions. But to achieve state-of-the-art performance, last generation's DPU hardware is not capable enough and tighter coupling between the DPU's hardware and remote storage server is required. The DPFS++ framework and its artifacts are publically available at `https://github.com/IBM/DPFS`.

Preface

This Master's thesis has been a long time in the making (one and a half years!). In April of 2022, I moved to Zürich for five months for a research internship at IBM Research Zurich. Together with Jonas and Radu, I explored the undocumented `virtio-fs` feature Jonas had found by chance in the NVIDIA BlueField-2 DPU. A year later, in March 2023, we built a prototype of our DPFS framework and wrote a research paper with Animesh. In June of 2023, I was given the opportunity to present our research paper and work at SYSTOR'23 in Haifa, Israel, something that I would never have imagined possible when I started with my Master's degree at VU Amsterdam in 2020.

I want to thank Jonas Pfefferle and Radu Stoica from IBM Research for allowing me to do research in the Hybrid Cloud Infrastructure Software group at the IBM Research Zurich laboratory, giving me an environment to learn about every layer of the Cloud hardware and software stack, and for helping me improve my research, presenting, and writing skills. I would also like to thank Animesh Trivedi, my supervising Professor from VU Amsterdam, for taking me on as a research student to research file systems and sparking an interest in storage systems research that has led to the work on DPFS and this Master's thesis.

I am incredibly proud to present the DPFS framework and its DPFS++ iteration in this Master thesis. So, without further ado, let's dive into the mysterious world of data centers, Cloud storage services, and DPU-offloading hardware.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

Our 21st-century society is built on top of and is dependent on technology; it reaches into every aspect of our daily lives. In the Netherlands, it is estimated that data centers and the *Information and Communications Technology* (*ICT*) sector enable over 3.3 million jobs and contribute to 60% of the GDP (64). Data-driven systems are becoming prevalent and critical to our daily lives. It is estimated that by 2023, our society will annually generate one yottabyte (one billion petabytes) (60). This data is being generated from and consumed in computer systems in multimedia streaming (e.g., Netflix, Spotify), social media (e.g., Instagram, Mastodon), IoT and edge computing (e.g., smart-home devices), scientific simulations (e.g., protein folding and climate modeling), transportation (e.g., shipment routing and smart energy grids), and collaboration tools (e.g., Microsoft Office365, Google Drive). To handle all this vital data, modern storage systems must be in place to gather, store, and analyze this data in an efficient, reliable, and performant manner.

As a computer service provider, satisfying the increasing needs in computing and data is challenging. Building and maintaining infrastructure is expensive, time-consuming, and requires specialized technical expertise. Due to these challenges, the industry has widely adopted the Cloud computing model, where the underlying infrastructure is decoupled from the application (11). This decoupling allows data center operators to build, maintain, and optimize systems that efficiently provide the application's compute, storage, and networking infrastructure requirements. Major Cloud providers like Amazon AWS, Microsoft Azure, and Google Cloud provide cost-effective and state-of-the-art infrastructure for research institutions, businesses, and governments.

Building such state-of-the-art cloud systems is becoming increasingly complex due to the increasing demands from applications and tenants (i.e., users who rent cloud resources), and a shifting hardware landscape. Performance improvements can no longer be achieved

by merely waiting for faster hardware. Since the mid-2000s, single-core performance improvements have stalled, and hardware has become increasingly heterogeneous, leading hardware designers to introduce compute silicon that exposes different programming models (57). On the other hand, new storage technologies like *flash* provide orders of magnitude lower access latencies than traditional hard disks (107), and the networking bandwidths are still increasing year-over-year (97). These changes in the hardware landscape have spurred a movement of rethinking existing system designs and programming interfaces in the software ecosystem (144).

The recent boom of *Artificial Intelligence* (*AI*) has given the systems research community a new wave of workloads that require massive amounts of resources from our cloud infrastructure. *Large Language Models* such as *ChatGPT* and *LLaMa* that show sparks of *AGI* (*Artificial General Intelligence*) cost millions of dollars in infrastructure to train and push the limits of our current cloud systems. One of the bottlenecks identified for AI workloads on the hardware and software stack currently deployed in the cloud is storage (16, 103). Training a state-of-the-art AI model requires a massive data set to train, which can be on the order of hundreds of gigabytes to terabytes (23, 82, 143, 162).

1.1 The state-of-the-art cloud is DPU-powered

The trend of heterogenous computing has manifested itself mainly in the form of *compute offloading* to *accelerators* that work alongside the *host CPU*. Workloads are executed on the accelerator to improve performance and increase efficiency (57, 160). Such accelerators come in many forms, such as GPUs for graphical and matrix operations and ASICs for fixed compute functions. This thesis and research focuses on one specific type of accelerator called the *Data Processing Unit (DPU)* that combines a *Networking Interface Card (NIC)* (used to connect the CPU with the network) with specialized offloading capabilities for data center purposes.

By tightly coupling several kinds of offloading hardware with the NIC onto a single accelerator, several domains of data center computing that utilize the network can be offloaded to the DPU. Current commercial DPUs like the NVIDIA BlueField can (among others) offload network switching, network virtualization, block storage virtualization, cloud orchestration, and encryption/decryption. This varied set of infrastructure offloads allows CPU to be focused on the workloads of tenants, thus generating more revenue and centralizing the data center operator's infrastructure on the DPU (12, 47, 72).

Today, all major cloud providers publicly state that they use DPUs in some form. Amazon AWS has its own in-house DPU called Nitro (12), Google Cloud utilizes the Intel IPUs (33), Microsoft Azure has its own in-house DPU called Azure Boost (148) and is partnering with Pensando (125), IBM Cloud partners with Pensando (125), and Oracle Cloud deploys the NVIDIA BlueField (116). In academia, DPU offloading is a "hot topic", with published literature on offloading of data center infrastructure workloads that show that DPUs are effective in aiding host CPU workloads to reach networking line speed and saving host CPU cycles (31, 74, 90, 91, 142, 157).

1.2 DPFS - A DPU-powered file system virtualization framework

Cloud storage is commonly exposed to tenants in three forms: block, file, and object (i.e., key-value) (13, 32, 35). Block storage exposes raw, unformatted storage blocks, enabling precise control over data placement and facilitating high-performance, low-latency data access, making it well-suited for applications like databases and virtual machines. File storage presents data in a hierarchical file system akin to traditional file servers, serving purposes requiring structured organization and shared access, typically utilized for documents, media files, and application data management. In contrast, object storage abstracts data into objects with unique identifiers, storing them within a flat address space, rendering it ideal for large-scale, unstructured data storage, such as images, videos, backups, and logs. These three storage types cater comprehensively to diverse data storage needs in cloud data centers, ensuring tenants can select the most fitting option based on their specific requirements, whether necessitating high performance, structured file access, or scalable, unstructured data retention.

In today's cloud, file storage is typically offered via a *virtualization gateway* that is exposed over the network through a standardized protocol called *NFS (Networked File System protocol)* (6, 50, 61, 101), whose goal it is to abstract away the details of the cloud operator's internal *distributed file system* (*DFS*). The tenant uses the NFS client offered by their operating system and connects to the gateway over the network. The cloud operator translates the NFS file system operations in the virtualization gateway to their internal DFS, thus transparently offering file storage to the tenant.

The gap in the literature motivates the research on file system virtualization using DPUs, which led to the development of the DPFS framework. The research is published in a

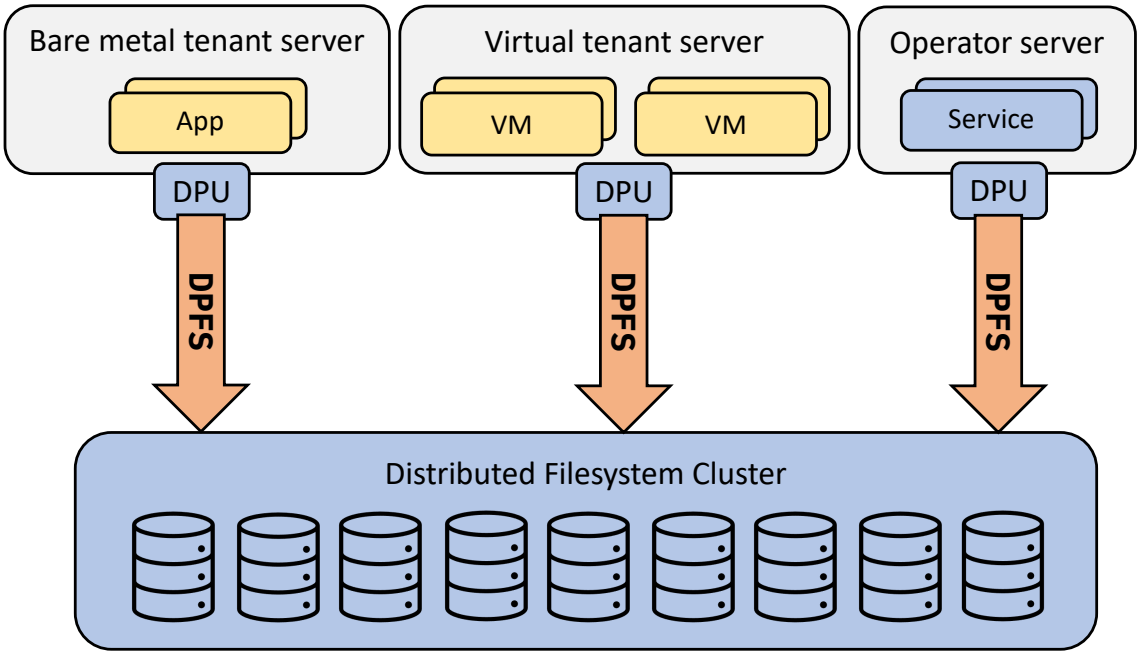


Figure 1.1: High-level architectural diagram of the DPFS framework and its proposed use in a cloud data center.

research article at the 16th ACM International Systems and Storage Conference (SYSTOR'23) in Haifa, Israel (51) (see Chapter 4 for an extended version of the paper). In the DPFS research, the authors propose to decouple the file system client from its networked protocol by virtualizing it with a commercial DPU using the Linux `virtio-fs` software stack. The decoupling allows DPFS to offload the file system client (that's connected to the remote storage system) execution to a DPU, which is managed and optimized by the cloud provider while freeing the host CPU cycles. The architecture of the proposed DPFS cloud storage system is shown at a high level in Figure 1.1. DPFS, the proposed framework, is 4.4× more host CPU efficient per I/O and 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.

1.3 Problem statement - Evolving DPFS into a cloud-native system

The DPFS paper proposes a novel file system storage design, showing that its efficiency surpasses that of NFS and that its small I/O performance is comparable to NFS. However, the paper has several weaknesses in its design, implementation, and experimentation that

prevent it from proposing a solution that cloud providers utilize (a classification often called cloud-native). These weaknesses are:

- **Lacks multi-tenancy** - DPFS only supports a single tenant on the host CPU. It can only expose a single virtual file system (i.e., `virtio-fs` device). At the same time, cloud environments virtualize a single CPU to support many tenants simultaneously.
- **Reliance on NFS gateway** - Ironically, DPFS can only expose remote storage to the tenant through an NFS gateway. This prevents DPFS from consuming existing cloud file storage infrastructure without the performance issues of utilizing a gateway.
- **Missing multicore evaluation** - The experimentation does not contain multi-core and large I/O size experiments. Workloads that are common in the cloud today, are massively parallel and operate on large data sets (29), so knowing the performance characteristics of DPFS under these workloads is vital.
- **Poor performance predictability** - The experimental results in the paper show that during heavy workloads, with many outstanding I/O operations, the latency variability of a DPFS operation increases significantly. This clashes with a fundamental goal of predictability that cloud operators strive for and are legally required to uphold via *SLA* contracts (11).
- **High overheads** - DPFS shows adequate performance compared to its NFS counterpart in the paper, however, we show the performance is far from what the underlying NVIDIA BlueField-2 hardware can provide.
- **Unknown hardware landscape** - The hardware landscape of DPUs is very diverse, with each DPU model having their own set of hardware characteristics, offload capabilities, and programming models. There is currently no work out there that systematically examines the DPU hardware and its capabilities for cloud data centers.

This thesis aims to fill these gaps in the design, implementation, and experimentation of DPFS by proposing the DPFS++ framework.

1.4 Research questions

With this problem statement in mind, we formulate the following four research questions that together aim to elevate DPFS towards a cloud-native file storage framework.

RQ1. How can the current generation of DPU hardware facilitate file system offloading?

This research question explores the current generation of DPU hardware and literature on distributed file system design using DPUs. The design space of DPU-offloaded distributed file systems is sketched by overlaying the currently explored designs and hardware. Allowing us to propose novel designs that have not been explored yet. One of these new designs will be implemented and evaluated in DPFS++.

RQ2. How can the performance of the Linux kernel be improved for DPU-offloaded file systems?

A cloud tenant consumes the DPU-offloaded file system provided by DPFS through the Linux kernel's `virtio-fs` device driver. We are aware of several limitations in the `virtio-fs` driver, that limit the multi-threading and performance capabilities of DPFS. This research question will first expand the ACM SYSTOR'23 paper with experiments and analysis so that the limitations of the `virtio-fs` driver and its effects on the DPFS framework become known. The research question will investigate the limitations of the `virtio-fs` driver, design improvements to the `virtio-fs` driver, and benchmark the implementation.

RQ3. How can DPFS efficiently utilize existing kernel-based distributed file systems?

DPFS currently supports two remote storage backends: NFS and Key-Value (i.e., translates file operations to a RAMCloud (120) key-value remote storage backend), which both implement the API that DPFS exposes and fully operates in the user space of the DPU's Linux operating system. Running in user space instead of interacting with the kernel has the upside of increased performance. However, this comes with the downside of needing to perform the manual software engineering labor of implementing support for a remote backend that a cloud data center might wish to use.

In this research question, we investigate the design, implementation, and performance implications of a new backend in DPFS++ that allows the DPU to expose any existing kernel-based distributed file system mounted in the DPU's kernel.

RQ4. How can DPFS efficiently support dynamic multi-tenancy?

With modern hardware, a single dual-socket server can contain up to 256 CPU cores (7). In a scenario where every tenant consumes only a single CPU in a virtual machine, there are already 256 tenants packed into a single server. Even if the server contains multiple DPUs, each DPU must still be able to provide its services to various tenants simultaneously. This is further complicated because cloud data centers are not static deployments. Cloud tenants

expect to be able to scale their consumption of cloud services to their needs dynamically. The current design and implementation of the DPFS framework only supports exposing a single `virtio-fs` device to the host (i.e., a single tenant) while entirely consuming a single DPU core. It is, therefore, evident that the DPFS framework must evolve to support multi-tenancy for such dynamic deployments efficiently. This research question will detail multi-tenancy support's design, implementation, and evaluation in DPFS++.

1.5 Research methods

The DPFS research project has been an experimental systems research project since its inception in April 2022. To answer the research questions, we will start by surveying the literature and hardware landscape to explore the design space of DPU-offloaded file systems (**RQ1**). Then, the performance characteristics of DPFS are analyzed through experimentation, performance engineering, and implementing improvements (**RQ2**). To drive DPFS towards a cloud-native system, we design and implement three functionalities in DPFS++: (**RQ3**) support for existing kernel-based distributed file systems like Ceph, (**RQ1**) support for the gateway design, and (**RQ4**) support for dynamic multi-tenancy.

The following computer systems research methodologies were used in the DPFS research project and this thesis. To survey the academic and industrial literature for related work on topics such as accelerator offloading and distributed file systems within computer systems research for **RQ1** and other parts of the thesis, the quantitative research methodology (85) was used.

The design and implementation of DPFS and DPFS (**RQ1, RQ2, RQ3, RQ4**) are conducted with the iterative and circular process of abstracting, designing, and implementing research ideas (53, 65, 124). By analyzing the requirements and behavior of the system, a design for the solution can be articulated and then implemented. When this leads to unexpected or undesired results, the cycle is repeated.

To identify bottlenecks (**RQ2**), measure the characteristics of the built system, and verify hypotheses (**RQ1, RQ2, RQ3, RQ4**), the method of experimental research (55, 66, 119) are used. This research method systematically conducts experiments and records system measurements via reproducible synthetic and real-world workloads.

Lastly, to make this work of research reproducible and accessible to the broader research community, the open science methodology (55, 149, 155) are applied. Everything legally permitted to be open source (including the framework, experimentation suite, and experimental results) is open-sourced at `https://github.com/IBM/DPFS`.

1.6 Thesis research contributions

This thesis makes the following seven scientific contributions:

C1: (Conceptual & survey) A deep-dive into current day cloud infrastructure (with historical perspective), DPUs and cloud storage systems - Chapter 2

C2: (Conceptual) A dissection of the DPU hardware landscape and how these DPUs form a design space for distributed file system offloading - Chapter 3

C3: (Conceptual) The design of DPFS++, an evolution of the DPFS framework including features for hybrid cloud deployment - Chapter 5

C4: (Dissemination) Three presentations on the DPFS project and peer-reviewed SYSTOR'23 paper (51)

- OpenFabrics Alliance'23 workshop presentation
- ACM SYSTOR'23 conference paper presentation
- CompSysNL'23 presentation

C5: (Experimental) An in-depth set of experiments on DPFS (complementary to the paper's experiments) and DPFS++, performed on the NVIDIA BlueField-2 - Chapter 6

C6: (Artifact) The implementation of the DPFS framework in its second iteration, including the improvements and features designed in this thesis -
source code: `https://github.com/IBM/DPFS`, detailed description: Chapter 5

C7: (Artifact) Reproducible experimentation suite with all the scripts and results - `https://github.com/IBM/DPFS`

C8: (Conceptual & Experimental) An expanded version of the original DPFS paper that adds several experiments and an investigation into the limitations of DPFS on the NVIDIA BlueField-2 - Chapter 4

The paper published and presented at ACM SYSTOR'23 titled *"*DPFS*: DPU-Powered File System Virtualization"*(51) is in itself not a contribution of this thesis, since it was worked on for the "Industrial Internship" and "Research Project Computer Science" courses. It is, however, included in this thesis in Chapter 4, in an extended form, as this thesis is a direct successor to the paper. The new contributions are marked with •.

1.7 Societal relevance

This research aims to design and implement a DPU-powered file system virtualization framework for data centers. Accomplishing this alleviates the following issues in today's data center systems.

Firstly, our society depends on and demands our internet and data center infrastructure. With increasing demand comes an increasing amount of data that needs to be stored, accessed, and processed in our data centers. It is estimated that the world's data will take up more than 200 Zetabytes by 2025 (i.e., 100 trillion gigabytes) (105) and that 20% of the world's electricity will be consumed by our digital infrastructure. This means that even marginally small inefficiencies in this infrastructure result in impactful monetary and environmental differences (64). Therefore, researching and optimizing data center infrastructure is vital to the goals of our society on the fronts of energy efficiency, sustainability, and e-waste.

Secondly, this work aims to be fully transparent towards data center tenants (i.e., companies who utilize data centers). Where other works of research surrounding DPU file storage require tenants' intervention, this research only needs to be adopted by data center operators. This lowers the barrier to adopting these advancements and reduces the software maintenance strain on data center tenants.

1.8 Plagiarism Declaration

I hereby confirm that this thesis is my own work, is not copied from any other source (person, internet, or machine), and has not been submitted elsewhere for assessment. I understand that plagiarism is a severe offense to the fundamental values of science and should be dealt with if found.

Note that Chapter 4 contains the contents of a paper (in an extended form) published and presented at the 16th ACM International Systems and Storage Conference (SYSTOR'23) in Haifa, Israel (51). The authors are Peter-Jan Gootzen (IBM Research and VU Amsterdam), Jonas Pfefferle (IBM Research), Radu Stoica (IBM Research), and Animesh Trivedi (VU Amsterdam). The new contributions added for this thesis are marked using • in Chapter 4.

2

Background

To help guide the reader through how DPFS came to be and how it will evolve in this thesis, this chapter will explore the motivation behind the advent of DPUs and their use in today's cloud systems. First, we examine the evolution of virtualization and hardware technologies that shaped and enabled cloud computing (Section 2.1). Second, we will explore how current hardware trends motivate a move towards offloading computing using heterogeneous architectures. In particular, we will focus on the offloading capabilities of networking interfaces, including DPUs (Section 2.2). Third, we introduce the new DPU-powered cloud model and identify the benefits that this new infrastructure model brings to a cloud data center (Section 2.3). Fourth and last, we analyze in detail the current day cloud file system services (which are not-DPU powered) (Section 2.4), exposing the virtualization limitations that DPFS aims to solve.

2.1 Cloud Computing

The advent of the Internet has brought about a significant increase in agile computing demands and a strong drive for reduced operating costs. Cloud computing has played a pivotal role in providing the necessary infrastructure (including hardware and software stack) to support the Internet-driven era of society. One of the fundamental attributes of cloud computing and cloud infrastructure is its ability to efficiently adapt to the dynamic resource requirements of tenants while maintaining cost-effectiveness (11). A key component to providing this elasticity to the cloud tenants (i.e., the users who rent resources in a cloud data center) is the technology and concept of *virtualization* whereby the hardware is abstracted and split into separate *virtual* pieces of hardware that tenants utilize.

2.1.1 A brief history of Virtualization technologies

In Computer Science, ideas and innovations are often revisited. Recent innovations in hardware, software, or theory often allow old ideas to be reborn differently. The concept of virtualization is one of these ideas. It has gone through many forms in the history of computing and is today an essential aspect of cloud infrastructure.

In the 1960s, IBM mainframes were virtualized by multiplexing the hardware between several users, each user having their virtual terminal (121, 136). This form of virtualization through multiplexing fell out of favor due to the increasing hardware complexity. The research community introduced a simpler virtualization alternative in operating systems like UNIX that employs a multi-user kernel that can time-share the resources (130). In the 1990s, there was another leap in virtualization technology. Process-based virtual machine approaches of virtualization emerged, which allowed users to further decouple their software implementation from the underlying hardware. For example, the Java Virtual Machine allows users to compile software into virtual machine byte-code, so that can be mapped at run-time onto the hardware (43, 126, 152). In the early 2000s, the hardware had become so increasingly powerful (Denard's scaling was still alive) that with novel and innovative designs, virtualization through multiplexing the underlying hardware became viable again with hypervisors like Xen (17) and VMWare (25). A hypervisor allows users to run their own full-fledged operating system with a kernel (a virtual machine) while maintaining adequate performance and isolation from other users. This virtual machine form of virtualization became the basis of modern-age cloud and data center infrastructure as we know it today (80, 139). The most recent evolution in virtualization technology came in the 2010s, with *containers*, that packages the application with its dependencies and configuration files into a singular deployment file. Many containers are deployed onto a single virtual machine by sharing the kernel, and virtualized by the operating system through *namespaces* and *cgroups*, which isolate the containers and their resources inside the kernel (122).

2.1.2 Modern Cloud Computing Infrastructure

The innovations in hypervisor-based virtualization and increasingly powerful commodity hardware (in terms of compute and networking) allowed data centers to adopt *Hyper-Converged Infrastructure* (*HCI*) that changed the data center's infrastructure to be software-defined (52). With HCI, the tenant-facing storage, networking, and security services are

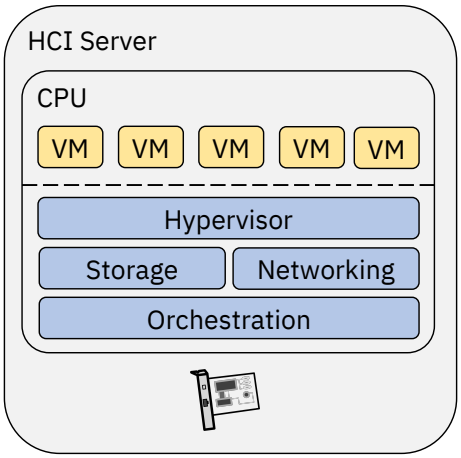


Figure 2.1: Architectural diagram of the hardware residing in a cloud data center server and the software systems running on that hardware.

software-defined and run on the same CPU as the tenant's compute alongside the hypervisor. This data center architecture is shown visually in Figure 2.1. In this diagram and others in this thesis, the yellow boxes indicate systems that are controlled by the tenants of the data center, and the blue boxes indicate systems that are managed by the data center operator. Allowing the operator to virtualize their internal services (i.e., not tenant-facing) to the tenants via the software-defined layer running on the CPU. The operator's internal services (i.e., not tenant-facing) are software-defined on commodity hardware and utilize networking hardware that supports software-defined networking. This model allows the datacenter operator to build a complete datacenter from commodity hardware, streamline the scaling of resources, and ease the management of services across the datacenter (37, 46, 80).

2.1.3 Limitations of the Hyper-Converged Cloud Infrastructure

In this work, we identify four key problems with the HCI approach for cloud computing concerning tenant-facing services:

Resource consumption Placing cloud services (storage, networking, and orchestration) next to the hypervisor, and thus on the host CPU, results in fewer host resources being available for the virtual tenants. This is economically suboptimal, as cloud operators mainly rely on selling CPU time for their revenue (47, 100).

Isolation A single point of security failure arises from the consolidation of virtual tenants and services. Suppose one of the virtual tenants breaks out of their respective sandbox (i.e., virtual environment). In that case, they do not only gain access to the other tenants

but also to the operator's internal services. With an increased focus on security in today's world, coupled with the recent emergence of side-channel attacks that can break out of the sandbox (75, 77, 89), the operators and tenants are not willing to accept the poor isolation that HCI provides anymore.

Bare metal support A bare metal tenant has complete control over the host CPU, and the operator cannot force the tenant to run any code on the host CPU. The operator cannot run their virtualized services alongside the tenant, and thus, the tenant cannot consume them. This leads to increased complexity for both the operator and tenant as they have to consider two distinct sets of services, one for virtual machines and one for bare metal machines (104, 164).

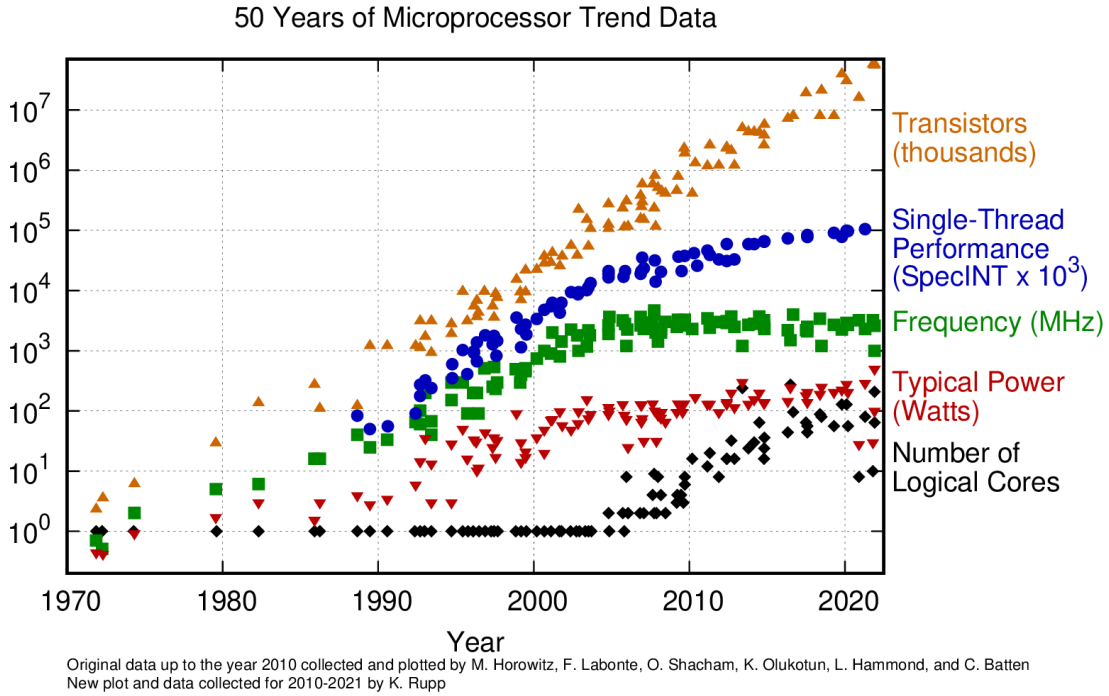


Figure 2.2: 50 years of microprocessors trend data, by Karl Rupp (132), permission obtained via `https://github.com/karlrupp/microprocessor-trend-data/blob/master/LICENSE.txt`.

2.2 The great wave of heterogeneous architectures

In recent years, everyone in the computing business has heard the claim "Moore's law is dead!". While this claim is, in essence, correct, it is technically not valid. Moore's observational law states that the number of transistors doubles every two years and is still in full effect in 2023. What has died is the scaling of single-core performance, which used to be scaling alongside Moore's law until Denard's scaling broke down around 2005 (57). The fall of Denard's scaling started the era of parallel and multi-core processing. This can be seen in effect in Figure 2.2, where around 2005, the steady increase in clock speed started to slow down, and the number of cores increased.

2.2.1 Offloading in heterogeneous systems

The wave of multi-core processors came in many forms, most commonly the symmetric multiprocessor (e.g., an x86 Intel/AMD CPU) and later cache-coherent non-uniform memory access (i.e., a coherent group of CPUs with differing main memory latencies), which became dominant in the commodity CPU market (56). But also a new proliferation in specialized hardware targeted for offloading compute-intensive workloads, freeing up the CPU and increasing performance per watt (22, 160).

However, this new form of parallel chip design is an entirely different beast to extract performance from compared to the single-core chips of before. Compute tasks must be spread around the cores to gain performance speedup, requiring specialized concurrent programming. The picture is even further complicated when looking at CPUs and domain-specific architectures (DSAs) that tailor the hardware to constrained forms of computation (i.e., not general purpose). To adjust the software to these DSAs, domain-specific languages (DSLs) have emerged alongside the new hardware (57). For example, the CUDA DSL allows a programmer to write massively parallel computational code that runs on NVIDIA GPUs (68), and the TensorFlow DSL allows modern neural networks to map efficiently to Google's TPU hardware (67, 68).

DSAs are commonly deployed alongside a CPU that gathers data and tasks from the storage and network to execute on the DSA hardware. This system model increases the computational performance and falls under the umbrella term of *offloading* (22).

2.2.2 A trajectory of increasing offloading capabilities in NICs

The networking hardware space has also seen the influence of DSAs and offloading. Networking Interface Cards (NICs) attached via an interconnect like PCIe have seen several

waves of offloading capabilities that increasingly offload traditional CPU tasks to the NIC. The naming scheme that vendors have adopted for these advanced NICs has been very inconsistent, with the terms DPU, SmartNIC, and Infrastructure Processing Unit (IPU) often being used interchangeably (135). In this thesis, we separate the waves of increasing offloading capabilities in NICs into four types of NIC hardware: the foundational NIC, the offload NIC, the SmartNIC and the DPU (73).

Foundational NIC The first type is the *foundational NIC*, which regards a bare-bones NIC without any offloading capabilities. These NICs are increasingly rare in the commercial market but can still be found in the embedded market. As there, the networking throughput requirements are in the order of gigabits per second compared to the hundreds of gigabits per second in the server market.

Offload NIC The second type of NIC is the *offload NIC*, which contains fixed-function offloads for popular protocols such as TCP and IP. These fixed-functions offloads aid the CPU in achieving line-speed networking and leaving computational room on the CPU for the workload.

SmartNIC The third type is the *SmartNIC*, which exposes programmable offloading capabilities through a DSL to the CPU instead of the fixed-function offloads of offload NICs. SmartNICs are aimed at the data center and cloud deployments, as the programmable nature lends well to the tight integration of the data center hardware and software stack controlled by the operator. A common offload capability of SmartNICs is the offloading of software-defined networking and network switching via *Open vSwitch* and *Open Virtual Network* (114, 140). By offloading these workloads off the CPU (traditionally running alongside the hypervisor) and onto the DSA hardware in the NIC, more CPU cores are freed up to be utilized by tenants, resulting in higher revenues for the data center operator (47, 87, 94, 164).

Data Processing Unit (DPU) The fourth and most offload capable type of NIC is the *Data Processing Unit* (*DPU*), which further increases the programmability of a SmartNIC through more general purpose DSLs, complete CPU cores or an FPGA element. Just like SmartNICs, DPUs are aimed at data center and cloud deployments but additionally become a full-fledged node in the data center (135) that runs operator software to orchestrate tenants running on the host CPU and to provide services to said tenants on the host CPU (12, 27, 73).

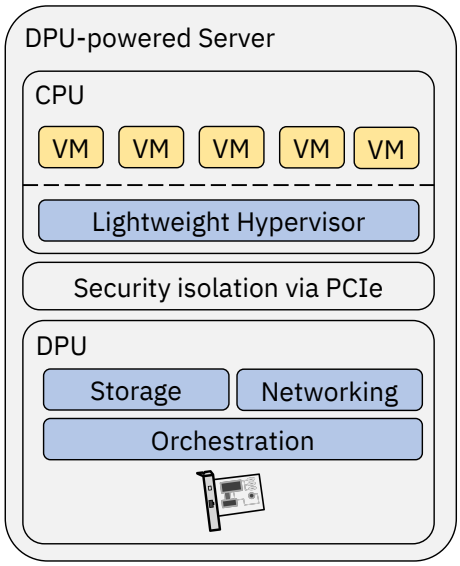


Figure 2.3: Architectural diagram of the hardware residing in a DPU-powered server and the software systems running on that hardware. Blue boxes indicate systems that the operator manages.

2.3 A brave new DPU-powered cloud

Previously, as seen in Figure 2.1, the cloud provider would run a significant (in terms of overhead and security attack surface) software stack on the CPU alongside the virtual tenants. This software stack includes storage services, software-defined networking services, and orchestration services (e.g., management, monitoring, and security of the virtual tenants).

Amazon AWS, the world's largest cloud provider (129), has deployed its custom-built AWS Nitro DPUs for all EC2 VPS instances since 2018 (12). Figure 2.3 shows an architectural diagram of how DPU-powered clouds like AWS today, are utilizing the DPU for offloading their cloud services. The cloud provider's services of storage, networking, and orchestration, shown in blue, are now running on the DPU and exposed via PCIe to the tenants running their workloads (e.g., virtual machines) on the host CPU.

The other large public cloud providers like Azure, Google Cloud, IBM Cloud, Oracle Cloud, and Alibaba Cloud now also publicly state that they are utilizing custom-built DPUs inside their data centers (8, 33, 47, 87, 161, 164). For data center operators that do not have the means to develop their own DPU hardware and software-stack, hardware vendors have started partnering with data center platforms to provide an off-the-shelf DPU stack. For example, NVIDIA and AMD Pensando are partnering with VMWare to

integrate the NVIDIA BlueField (27) and Pensando Distributed Services Card (DSC) (97) into the VMWare vSphere virtualization product (113, 125).

2.3.1 Alleviating the issues HCI faces using the DPU

The new DPU-powered cloud deployment model addresses the issues raised with the HCI Cloud in Section 2.1.3.

CPU consumption By running the services on the DPU, the host CPU is freed up to run the tenants' workloads, leading to reduced costs (86). This is especially important when the tenants are using software that is licensed (and paid for) on a per-core and utilization basis (100). Thus, effective CPU utilization additionally impacts costs (on top of the hardware and electricity costs).

Isolation Utilizing the DPU introduces an isolation barrier between the tenants running their code on the host CPU and the operator services running on the DPU. Take, for example, authentication keys that the operator uses to connect to their internal remote services. When running the operator services on the host CPU, these keys will reside in the CPU's caches and main memory, forming an attack surface with side-channel attacks. On the other hand, in the DPU deployment model, the keys will reside off-CPU on a PCIe-attached device, the memory of which the host CPU cannot access without the DPU's intervention (27, 104, 164).

Bare metal support With HCI, the bare metal tenant generally has access to fewer services, as the operator cannot run their services on the host CPU. With a DPU, the bare metal tenants can utilize the same scalable services as virtual tenants. This decreases the complexity not only for the tenant but also for the operator. The operator's set of services can be unified, thus reducing engineering and management complexity (104, 164).

2.4 Current state of file system services in the cloud

While much of the research literature on DPU-powered cloud storage focuses on block storage (99, 161), file system storage also still plays a vital role in the storage offerings of today's cloud providers alongside block and object storage. In modern workloads like machine learning and data analytics, the need for multi-tenancy access to data and the usage of the POSIX file system API are still prevalent and, therefore, require modern file system services (32).

2.4.1 Adoption of distributed file system research faces challenges in the cloud

Cloud file system services are an extension of the classical *distributed file systems* (*DFS*). A DFS strives to satisfy two main properties: high availability and redundancy of a user's file data. Common techniques to achieve these properties are replication (137, 154, 163) and erasure coding (1, 28, 39). Classical distributed file systems are not suited for cloud deployments where the tenants (i.e., users) must be securely isolated from one another, and tenants must be agnostic towards the cloud operator's distributed file system (i.e., no specialized software stack required for the tenant).

An example of state-of-the-art distributed file system research that is challenging to adopt in a cloud data center, is the LineFS DFS (74). While it provides state-of-the-art performance, consistency, and reliability, it is not suited for cloud deployments because it utilizes a software/hardware co-design inside the host CPU operating system in the form of a custom kernel module. Cloud tenants expect to be able to run their unmodified software on top of the infrastructure. They are not keen on integrating untrusted kernel code into their operating system because of its security implications (150).

2.4.2 Virtualizing distributed file systems

To solve the issues of directly deploying a distributed file system in a cloud data center, the cloud operator virtualizes their DFS by deploying a gateway layer (6, 50, 61, 101). This gateway isolates the tenant from the DFS network by translating a common networked file system protocol such as NFS (54) to the DFS protocol while ensuring security (2).

Figure 2.4 shows how the gateway sits between the tenant and the various subsystems of the DFS. In the case of the traditional DFS, the tenant (shown in yellow) directly communicates with the DFS (shown in blue) using the DFS's specialized communication protocol and software stack. Where-as, a virtualized DFS is connected to using off-the-shelf software stacks provided in the operating system (e.g., Linux with NFS over TCP) via a *gateway* (i.e., the virtualization server).

The networked virtualization has significant downsides related to performance, availability, load balancing, and cost (51, 87). A downside that isn't fully articulated in the literature is the issue of standardization. Using standardized file system protocols on the tenant side is advantageous for ease of use. However, it severely stifles innovation. The NFS protocol, for example, is standardized through IETF's RFC process. This process has served the Internet and its interoperability well, but it is not optimal for self-contained

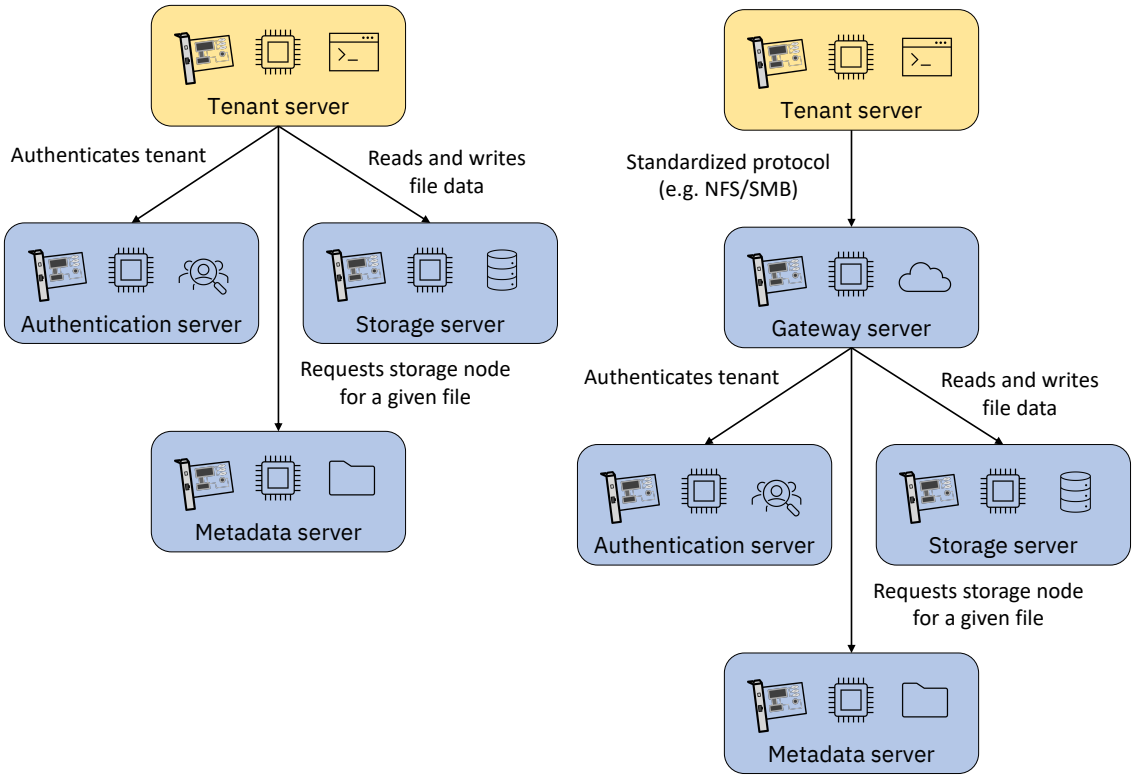


Figure 2.4: High-level architectural diagram of a traditional distributed file system (left) and a cloud file system service (right)

and quickly innovating systems such as cloud data centers. There have been revisions in the NFS protocol to add features that aim to keep up with the increasing performance of storage and networking hardware (58, 109). One such prominent NFS protocol feature is RDMA transport support (106), which seeks to alleviate the server CPU bottleneck by allowing the client to directly read file data from the server without involving the server CPU. Tenant-facing services in a cloud environment are generally not provided over the RDMA transport as it faces many scalability and isolation challenges (15, 49, 78, 153). Today's leading cloud providers, therefore, do not offer support for RDMA in their file system services at this point and are thus stuck with using outdated the NFS over TCP/IP protocol.

3

The design space of offloaded distributed file systems (Related work)

Having established the general cloud environment in today's computing landscape and the challenges distributed file systems face for the cloud in Chapter 2. An opportunity arises for integrating the DPU into the domain of file system services inside the cloud data center. Today, many cloud services are already being successfully offloaded on a DPU (12), and there is an opportunity for the DPU to integrate into the file system services of the cloud data center. While several papers present specific distributed file system designs that utilize a DPU, none of these works make a concerted effort to explore the hardware and file system "design space".

In this chapter, we explore the current state of DPU hardware (Section 3.1) and the currently explored design space of utilizing DPU hardware for designing a file system service (Section 3.2). Using this overview of the studied design space, we identify gaps in the design space and propose three novel designs (Section 3.3).

3.1 State-of-the-art DPU hardware

To systematically analyze the available design space for file system offloading using DPUs. We first closely examine the current hardware offerings public as of August 2023. In line with the dissection of the levels of offloading capabilities found in NIC hardware from Chapter 2, we include all hardware that we classify as a DPU (even if the manufacturer

calls it a SmartNIC). To compile the complete list of DPU hardware used in this analysis, the following three methods are used:

- The academic literature on DPUs is studied.
- The use of DPUs by the cloud providers (which we define as Amazon AWS, Microsoft Azure, Google Cloud, Oracle Cloud, IBM Cloud, Alibaba Cloud, and Tencent Cloud) was studied.
- The World Wide Web is searched for commercially available hardware offerings.

For this analysis, we divide the hardware into four categories based on how they allow the operator to program the offloading capabilities, namely FPGA (Section 3.1.1), generic CPU cores (Section 3.1.2), DSA (Section 3.1.3) and fixed-function ASIC (Section 3.1.4). There is an overlap between these categories as most of the DPUs contain generic CPU cores next to an FPGA, DSL, or fixed-function ASIC offloading engine(s).

	ASIC	**DSA**	**FPGA**	**CPU**
Programmability	None	Low	High	High
Programming complexity	n/a	Medium	High	Low
Energy efficiency & performance	High	Medium	Medium	Low

Table 3.1: Hardware specifications and software versions of the machines (including the DPU) used in the evaluation of DPFS++.

3.1.1 FPGA

The first category of offloading capabilities found in DPUs is the *Field-Programmable Gate Array* (*FPGA*), a particular type of integrated circuit that is reconfigurable and more performant than a general CPU. An FPGA is programmed using the same descriptive circuit language (called an *HDL*) as an *ASIC* (a "set-in-stone" integrated circuit). Still, because of its physical properties, it can be reconfigured after deployment in a system. This reprogrammability capability of FPGAs comes at the cost of lower performance and increased power consumption compared to ASICs (20, 81, 98). Still, it allows an FPGA to be useful for applications in scenarios where the non-reconfigurable nature of ASICs are not viable, for example in artificial intelligence (24), industrial machinery (127) and high-frequency trading (84).

There are currently three commercially available DPUs that contain an FPGA, namely the Intel IPU (26, 104), the AMD (previously Xilinx) SmartNICs (9, 40), and the Achronix

VectorPath (4). On the non-commercially available side of the DPU hardware landscape, we find that all the in-house DPUs by the cloud providers, except for Amazon AWS (i.e., Google Cloud, Microsoft Azure, Tencent Cloud, and Alibaba Cloud), contain an FPGA as their main offloading engine (26, 34, 47, 94, 164). Their tight vertical integration in the software and hardware stack, and amount of technical expertise, diminishes the downsides of an FPGA-based approach (i.e. more complex to program than CPU or DSL offloading).

3.1.2 Generic CPU cores

A commonly used framework for dividing the tasks performed in a data center is the *data plane* and *control plane* model. The control plane is the part of the system that executes ad-hoc or planned management tasks like updating software, handling configuration changes (by the operator or tenant), handling errors or special edge cases, and orchestrating the data center in general. The other part of the system is the *data plane*, which is optimized for low latency and high throughput. The data plane generally supports all the packets and operations that applications produce or trigger, to support the high performance needs of tenants (158).

Generally, DPUs contain generic CPU cores to run the control plane of the data center's infrastructure (12, 26, 40, 41, 96, 111, 112, 138, 140). Most of the current generation of DPUs use Arm cores for this purpose, likely because they are widely supported in Linux and higher levels of the software stack. By being able to run a Linux-based software stack on the DPU for control plane related purposes, data centers can more easily adopt DPUs, than when only having specialized hardware (like an FPGA or DSL offloading engines) on their DPU.

Most DPUs let the operator offload arbitrary computational tasks off the CPU and onto the Arm core(s) and Linux software stack. The Broadcom Stingray takes this approach to the extreme and only has the Arm cores for programmable offloading (21).

3.1.3 DSL offloading engines

The third category of DPU hardware offloading we define is the *Domain Specific Language* (*DSL*) offloading engine; this is not one specific type of programmable offload, but a set of programmable offloads that each have their different properties and goals (57). In general terms, every DSL offloading engine focuses on a specific domain of computing to improve performance compared to a generic CPU. This comes at the cost of needing to program for a specific type of DSL offloading engine.

Datapath accelerator

The most common type of DSL offloading engine is the *datapath accelerator*, which is included on the NVIDIA BlueField-3 (112), Fungible DPUs (138) and Kalray DPUs (41, 71). These engines allow the operator to write offloadable code in the C programming language and compile them against their specific instruction architecture (NVIDIA and Kalray do not specify their architecture, and Fungible uses MIPS) using a supplied *Software Development Kit* (*SDK*). An offload in this context is any computational function with a trigger event and an output event. An offload triggers based on hardware events, e.g., an incoming network packet or signal from the host CPU, and finalizes by optionally sending a packet over the network or replying to the host CPU. Specific capabilities of the datapath accelerator vary per DPU manufacturer and model.

P4

Two commercially available DPUs (the Pensando DPUs (97) and Netronome SmartNICs (140)) support executing programmable offloads through the P4 programming language. The P4 language is designed for processing packets at line speed on networking hardware, where the available resources of the hardware, limit the definition of processing. The language is Turing complete but constrained in some aspects, such as not supporting infinite loops. At compile-time, the programmer knows whether their processing pipeline will "fit" onto the hardware (19).

GPU

Lastly and possibly most uniquely, the NVIDIA BlueField line of DPUs also has models with a top-of-the-line GPU onboard that NVIDIA calls *Converged Accelerators* (115). These DPUs put Arm cores, a NIC, a GPU, and other offloading engines onto a single PCIe card. The GPU is accessible from the Linux operating system running on the Arm cores and must be programmed using NVIDIA's GPU-specific programming language *CUDA*. Having the GPU and NIC on a single PCIe card allows the GPU to be directly accessible over the network without the intervention of a traditional x86 host CPU. NVIDIA claims this removes the host CPU bottleneck for networking in GPU computationally heavy workloads such as distributed artificial intelligence.

3.1.4 Fixed-function offloading engines (ASIC)

Where the previous three offloading hardware categories were in some form and to some extent all programmable, the fourth and last category we identify is the set of *fixed-function* offload engines. As the name implies, the function (input, computation, output) is not programmable and fixed in the chip (often as an ASIC). The specific functions differ widely between DPU models; however, they generally represent the three service subdomains of a data center: security, storage, and networking. Examples of fixed function offloads commonly found in commercial DPUs are encryption/decryption (e.g. AES, RSA, or IPsec) (26, 26, 40, 71, 111, 112, 138, 140, 140), compression/decompression (e.g. deflate or LZ4) (26, 26, 111, 112, 138), data filtering (e.g. using regex) (111, 112, 138), NVMe to NVMe-oF translation for remote block storage (26, 26, 40, 71, 111, 112, 138), and Open vSwitch (i.e. configurable network switching) offloading (26, 26, 40, 71, 111, 112, 138, 140, 140).

3.2 The current DPU-offloaded file system design space

This section examines the current literature on distributed file storage offloading using DPUs (inside and outside a data center environment). We are unaware of any publicly available industrial or commercial systems for distributed file storage offloading using DPUs. Surveying the literature, we find that there are currently three academic papers on distributed file system offloading to DPUs (44, 74, 87). To depict the currently explored design space, we categorize these three academic research systems into two designs: *partial offload* and *guided passthrough*.

3.2.1 The partial offload design

Distributed file system tasks traditionally executed on the host CPU by the DFS client, such as the replication of data across other nodes (to prevent data loss in case of a node failure), take up significant host CPU resources and cause performance bottlenecks and instability. In the *LineFS* paper (74), the authors propose a DFS client that offloads several of the CPU-intensive DFS tasks to the DPU. Their experiments show that by tightly integrating the host DFS client and a commercial DPU's software stack, and executing the noisy DFS tasks on the DPU instead of on the host CPU, application latency is improved by up to 80%.

The second research system that we categorize as a partial offload design was proposed by Di Girolamo et al. at SuperComputing'22 (44). Like LineFS, the system we refer to as *Girolamo* also integrates the host DFS client and DPU software stack but takes a more transparent angle towards the host DFS client. In Girolamo, the host DFS client only performs the fundamental DFS operations (i.e., communicating with the metadata server and storage nodes) on the host CPU and tasks the DPU to perform the DFS *policies* on a per-operation basis. The DFS policies include noisy tasks like replication and security-related tasks such as authentication and authorization. This removes the need for the host CPU tenant to be aware of the DFS operator's policies. Making it possible for the DFS operator to modify the policies without requiring access to the tenant's system or requiring the tenant to upgrade their DFS client.

3.2.2 The guided passthrough design

The second design type that we find in the literature is the *guided passthrough* design, deployed in production cloud data centers by Alibaba since 2020 in their *Fisc* system (87) with their in-house FPGA-based X-Dragon DPU (164). In Fisc, the amount of work performed by the DFS client on the host CPU and the DPU is minimized through tight software and hardware co-design, and the DPU passes through the file system operations to the remote backend in a *guided* manner. The DPU contacts a *proxy server* (every storage node is co-located with a proxy server) to learn which storage nodes hold the data of a chunk of a file. This data is cached inside the DPU for future requests. The choice to minimize the work performed by the DPU is rooted in the resource constraints imposed by an FPGA-based DPU that is also utilized for other cloud systems and services.

3.3 Opportunities in the design space

In this section, we identify the gaps in the current design space (Section 3.2) by changing design parameters to create new designs and identify hardware (Section 3.1) that can power these designs. We propose three designs currently left unexplored in the literature: full offload (Section 3.3.1, gateway passthrough (Section 3.3.2), and decoupling (Section 3.3.3). These three novel designs are used in the design of DPFS in Chapter 4 and DPFS++ in Chapter 5.

3.3.1 The full offload design

The three research systems described in the academic literature perform all distributed file system computations and logic either partially on the DPU and partially on the host CPU (in the case of LineFS and Girolamo) or entirely on the remote server (in the case of Fisc). When we take this design parameter to an unexplored extreme, we find a design where the DFS is fully offloaded onto the DPU. In this design, all elements of the DFS client are executed on the DPU, and the operating system running on the host CPU passes all file system operations to the DPU and delegates the resolving of the operations entirely to the DFS client running on the DPU. Currently, DFS clients are written as software for generic CPUs; implementing this design could leverage the common ARM-based DPUs (such as the NVIDIA BlueField) to run said DFS clients. FPGA-based DPUs could also be leveraged by utilizing *high-level synthesis* (*HLS*) programming frameworks that FPGA vendors offer that allow "high-level" C or C++ code to run on an FPGA with minimal changes.

This design approach is implemented in DPFS by running the full DFS client on the ARM cores of a NVIDIA BlueField-2 and is further described and advocated for in Chapter 4.

3.3.2 The gateway passthrough design

The Fisc paper shows that a guided passthrough design can provide high throughput and low latency in a cloud multi-tenancy environment. However, we identify a critical drawback that can be alleviated by drawing inspiration from the NFS gateway architecture currently deployed in many cloud data centers (see Section 2.4.2). Namely, the guided passthrough design requires the remote storage backend to be tightly integrated into the software stack that runs on the DPU. This prevents the implementation of said design from being portable between different deployments of independent data center operators, where many different storage backends are in use (see Section 4.4.3). This contrasts with the NFS gateway design, whose portability between various remote storage systems has contributed to its widespread use in today's Cloud data centers.

The second design that we propose in the design space of DPU-offloaded DFS virtualization systems is the *gateway* passthrough design, which combines the guided passthrough design proposed by Fisc and the system design that a DFS with an NFS gateway implements (see Section 2.4.2). The gateway design is visually shown in Figure 3.1. In this design, when the DPU receives a file system operation from the tenant's operating system

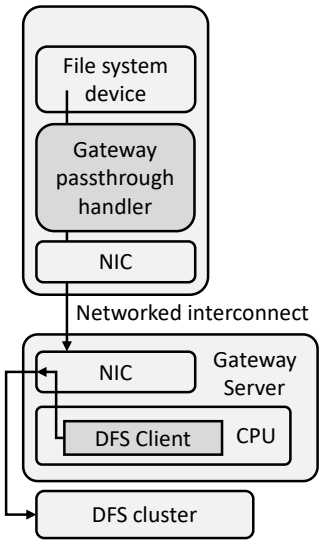


Figure 3.1: Diagram of the gateway design

on the host CPU, it is immediately forwarded to a remote gateway server. The gateway server is tasked with resolving the operations by running the DFS client on said server. This allows any currently deployed DFS to be reused in a DPU-based system (i.e., no integration in the remote side of the DFS), and compared to the NFS gateway approach, the advantages of a DPU-centric approach like the decoupling of the network stack and improved maintenance capabilities are retained.

DFS clients take up a considerable amount of hardware resources under demanding workloads (74), and the resources found in current-generation DPUs are meager compared to those found in current-generation server CPUs (quantified in Section 5.3. An advantage of the gateway passthrough compared to the full-offload design, which can also reuse any existing DFS client, is that it is not limited by the resource constraints imposed by the DPU hardware. As the DPU is only tasked with passing through the operations from the host CPU to the remote gateway, the available bandwidth of a gateway passthrough implementation is expected to be significantly higher than that of a full-offload implementation.

The gateway passthrough design is implemented in DPFS++, and the implementation is described in Section 5.3 and evaluated in Section 6.4.

3.3.3 Decoupling of the file system client on the host and the DPU

Tight software and hardware co-design across multiple layers of the stack has been a widely successful system design method in academic research and industrial data center deployments (79, 131, 145). However, as touched on in Section 2.4.1, this co-design can

make it difficult for cloud data centers to adopt these innovations for tenant facing services. The three systems in the current design space (LineFS, Girolamo and Fisc) also face this issue, requiring a custom DFS client integrated with the other software and hardware stack layers.

To remedy this issue, a standardized file system over PCIe protocol like `virtio-fs` (147) can be used to let the DPU expose itself as a file system to the tenant. This protocol is already upstreamed in the Linux kernel and requires no kernel or userspace software modification on the host's CPU software stack. With such a standardized protocol, the host file system client's implementation is decoupled from the DPU. It is left up to the DPU to decide how much of the DFS's tasks are performed on the DPU (e.g., the full-offload design) or a remote server (e.g., the guided passthrough design or gateway design).

Several commercial DPUs can expose arbitrary PCIe devices to the host CPU that the operator on the DPU programs. For example, the Pensando DPU (97) offers a SDK that can be programmed from the P4 engine or the Arm cores, which allows one to interact with the PCIe hardware and create a PCIe device from P4 or C code. FPGA-based DPUs, like those from AMD (previously Xilinx) (9), also have similar SDKs that can be utilized to expose a PCIe device (and thus a `virtio-fs` device) to the host CPU.

The decoupling design is an essential part of the design of DPFS, which is further detailed in Chapter 4.

3. THE DESIGN SPACE OF OFFLOADED DISTRIBUTED FILE SYSTEMS (RELATED WORK)

4

DPFS - a DPU-Powered File System virtualization framework

The design space explored in Chapter 3 introduced several design options for the manner in which the host CPU, DPU, and the remote storage server interact and how they divide their responsibilities. As a first effort towards evaluating parts of the design space that previously have been unaddressed in the literature, we present the DPFS framework that presents the design, implementation, and evaluation of a full-offload design using the NVIDIA BlueField-2 (111). We report that using the DPFS framework, the host CPU is 4.4× more efficient per I/O, and 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`.

This chapter consists of an extended version of the following paper: "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). Association for Computing Machinery, New York, NY, USA, 1–7. `https://doi.org/10.1145/3579370.3594769`". The new contributions that were added for this thesis are marked using • in this chapter.

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

4.1 The case for a full-offload DPU-Powered File System Virtualization framework

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 (107, 146). As a result, delivering the full speed of I/O devices in a disaggregated storage setting takes a considerable amount of CPU resources (76, 145). For example, Alibaba reports that 12 CPU cores are required to deliver 200 Gbps of block-level traffic (99). 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 (74). The question of CPU efficiency is also important for bare-metal machines, which have become popular in clouds recently (42, 118, 161). 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 (5, 87). 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 (108) or InfiniFS (95) 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.

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, lightweight, 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 (88), scalable metadata lookups with KV stores, decoupling of data from metadata management (83). This virtualization design combines the full-offload and decoupling designs proposed in Section 3.3.

A limited form of such decoupling is currently offered by cloud providers in the form of an NFS gateway to the CNFS client (6, 50, 101). 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 (Section 4.3.4). 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. Block-level offloading allows a fully offloadable I/O stack (86, 102), however, it suffers from significant I/O amplifications. We quantify this amplification (93) in Table 4.1, 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 the 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 (42), thereby improving tenant isolation and security.

In this chapter, we present DPFS, a DPU-Powered File System virtualization framework for cloud environments, based on the full-offload design from Section 3.3.1 and the decoupling design from Section 3.3.3. DPFS leverages the `virtio-fs` protocol to communicate between the host and the DPU. The DPU and host communicate via PCIe memory-mapped `virtio` (133) queues using the FUSE file I/O command set. As `virtio-fs` is standardized (147) and included in the Linux kernel (128) (no installation required), this design enables DPFS to avoid the need for running a custom CNFS client on the host. The *decoupling* of the front-end (a standard file system API) from its backend implementation

(i.e., the CNFS) with a bump-in-the-wire architecture on the DPU (47) 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) (117); (iii) `DPFS-KV`, a RAMCloud-based backend that implements low-latency I/O operations on small files, a suitable fit for KV stores (120).

The contributions in this chapter 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 (111) (`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.

4.2 The design and implementation of DPFS

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 4.1. 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* (3)) on top of the raw `virtio-fs` protocol buffers (`DPFS-FUSE` Section 4.2.3). 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` (Section 4.2.4), an optimized userspace

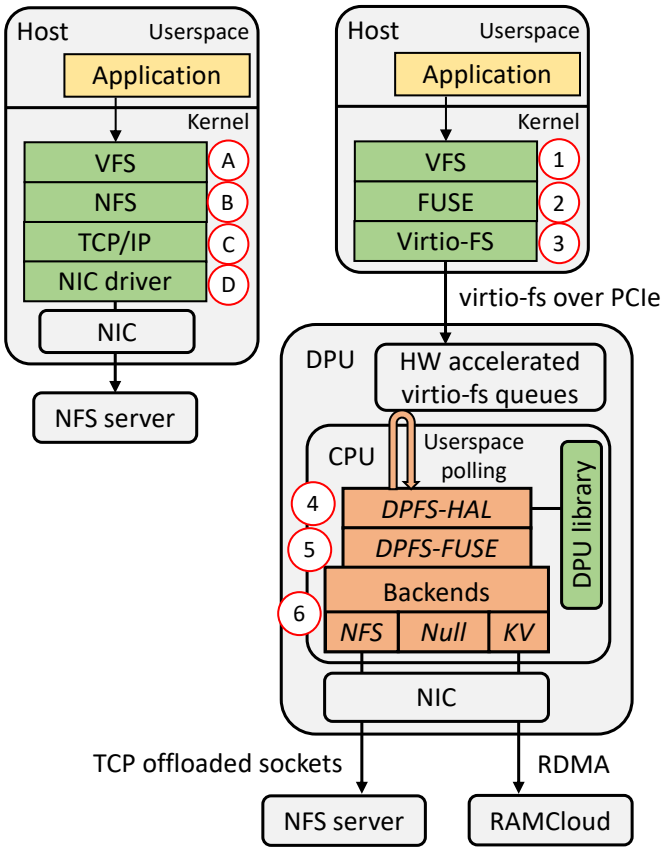


Figure 4.1: The architecture of DPFS compared to a host NFS client. Green boxes are the standard in-kernel code, and orange boxes are our contributions.

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` Section 4.2.5) that connects to a RAMCloud (120) 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 (79).

4.2.1 Design constraints of DPFS●

Every system is designed under certain design constraints that affect the capabilities and characteristics of said system. The key design constraints of DPFS and the DPFS project come from the goal of forming a cloud storage service. Cloud providers require high-performance levels, efficiency, interoperability, and maintainability between their systems and those of the cloud tenants. We define the following five design constraints under which

DPFS is designed and how they influence its design:

- **No application changes** - In this work, we constrain ourselves to support all existing file-based workloads on Linux systems, namely those going through the *VFS* virtual file system of the kernel.
- **No host kernel changes** - Cloud tenants are not keen on changing their validated Linux kernel to include untrusted code supplied by a third party at the kernel's security level. Therefore, we only use Linux kernel functionalities already included in the stable releases, i.e., the `virtio-fs` device driver that can be accessed through the VFS.
- **Minimizing performance overheads** - To provide file storage performance that can keep up with increasing storage demands from modern workloads like AI (16, 103), we design the `DPFS-HAL` and `DPFS-FUSE` layers to perform minimal processing and the backends (`DPFS-NFS` and `DPFS-KV`) to utilize high-performance computing techniques like TCP offloading and RDMA.
- **Supporting existing cloud infrastructure** - To aid the transition of file system services in cloud data centers, we design DPFS to consume existing DFS infrastructure using NFS via `DPFS-NFS` (and kernel-based file systems in DPFS++ via `DPFS-Kernel` in Section 5.2).

4.2.2 The DPFS I/O Path

In the host NFS client configuration (see Figure 4.1, 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 4.1, 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 Ⓑ-Ⓒ 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 Section 4.3.4.

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 (151). 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.

4.2.3 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 (3), 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.

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

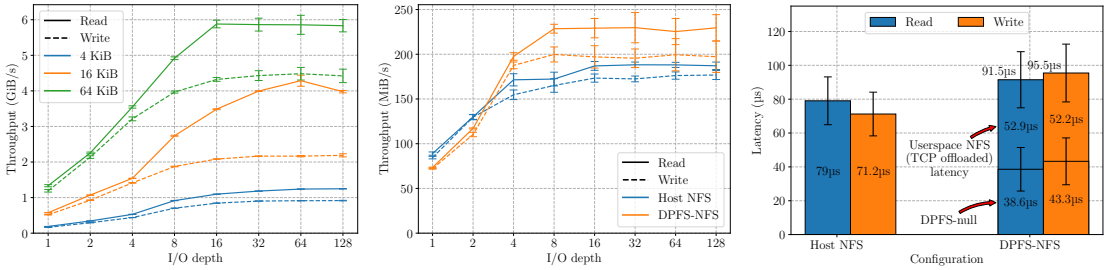
The `DPFS-NFS` backend implementation leverages the accelerated socket API of NVIDIA XLIO (117) and the *libnfs* library (134). NVIDIA XLIO is a shared library that overloads the socket C-API to leverage the network offloading capabilities in the BlueField-2 hardware with reduced data copies (not fully 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 (109) 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.

4.2.5 `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 (120). `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.



(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 4.2: Three DPFS performance experiments

4.3 Evaluation

In this section, we evaluate the performance and efficiency of DPFS through the `DPFS-Null`, `DPFS-NFS`, and `DPFS-KV` backends. Section 6.1 and Table 6.1 detail our experimental setup, where the *Gateway server* acts as our NFS server in this chapter. All workloads are generated using *fio* with the *io_uring* I/O engine (45). The `DPFS-Null` and `DPFS-NFS` throughput experiments that were in the original SYSTOR'23 paper (51) are reproduced from scratch on the new experimental setup of this thesis (see Section 6.1), the others are unchanged. The differences between the paper's experimental setup and this thesis's experimental setup are three-fold: (1) only a single fio thread is used instead of two (unless indicated otherwise), (2) single-threaded workloads are now pinned to the core on which the DPU's interrupt affinity is mapped, (3) multi-threaded workloads are now pinned to the DPU's NUMA node.

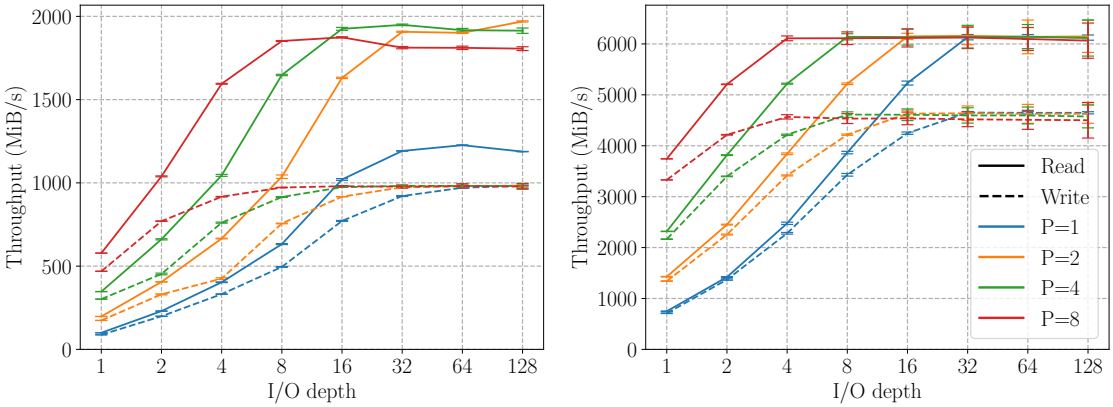
4.3.1 Virtualization Overheads (`DPFS-Null`)

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 4.2a shows throughput of the null-DPU backend (y-axis, in GiB/s) vs. the I/O (queue) depth. We report throughput for both read and write operations for 4 KiB,

16 KiB, and 64 KiB block sizes. 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.

Multi-threading



(a) `DPFS-Null` random 4 KiB I/O throughput **(b)** `DPFS-Null` random 64 KiB I/O throughput

Figure 4.3: `DPFS-Null` multi-threading experiment with 1,2,4 or 8 fio threads (P)

In Figure 4.3, the throughput of `DPFS-Null` is shown under a synthetic fio workload of 4 KiB and 64 KiB random I/O, where the fio thread count is varied with $P = \{1, 2, 4, 8\}$. The blue $P = 1$ line in Figure 4.3a is identical to the 4 KiB line in Figure 4.2a, and the three other lines in the figure ($P = \{2, 4, 8\}$) show how the system behaves under a multi-threaded workload. Note that the I/O depth of the x-axis is on a per-thread basis, so with 4 threads and an I/O depth of 32, the total system I/O depth equals $4 * 32 = 128$.

With this experiment, we find that for a small block size of 4 KiB (Figure 4.3a), a single fio thread cannot saturate the `DPFS-Null` device (its CPU utilization is 100%). When moving to two threads, the device can already be saturated (at I/O depth 32, and more threads do not result in a significant increase in performance. For the larger block size of 64 KiB (Figure 4.3b, we report that a single fio thread can saturate the `DPFS-Null` device

and that 1, 2, 4 and 8 fio threads all achieve the same performance for a certain total system I/O depth.

4.3.2 `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 4.1) 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), highlighting any inefficiencies.

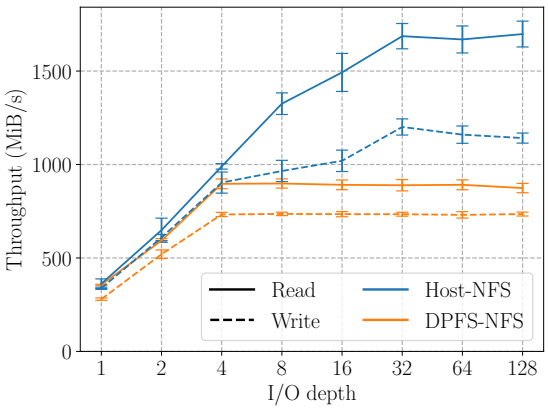


Figure 4.4: `DPFS-NFS` large block size experiment

Figure 4.2b shows the throughput (y-axis in MiB/s) vs. the I/O depth (x-axis) of 4 KiB random I/O. 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 an I/O depth of 16 (see limitations Section 4.4).

For a larger block size of 64 KiB, Figure 4.4 shows that the throughput of `DPFS-NFS` already maxes out at an I/O depth of just 4, while the throughput of host-NFS continues to increase until an I/O depth of 32. Resulting in the host-NFS system outperforming `DPFS-NFS` by 94% and 55% with a block size of 64 KiB for random reads and random writes, respectively.

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

Table 4.2: Host NFS vs. DPFS-KV 4 KiB file I/O latencies.

We further analyze the latency of 4 KiB I/O operations in Figure 4.2c. 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 also to 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.

4.3.3 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 4.2 shows our results of accessing complete 4 KiB files stored in NFS and RAMCloud (5 seconds warm-up, followed by a 10 second run). These files are accessed entirely (i.e., a single I/O operation). Compared to host NFS, DPFS-KV offers 7-12% latency gains over read and write operations.

4.3.4 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 misprediction 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

	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 4.3: The microarchitectural profile of the host CPU.

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 4.3 shows that (1) `DPFS-NFS` is light-weight compared to the host NFS, requiring 62.80% fewer 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 Section 4.4). 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).*

4.4 Limitations

The current work is limited by a few restrictions, which are addressed in DPFS++ (Chapter 5) and can be adressed with the advent of more powerful DPU hardware. In this section, these limitations are explored by examining the current state of DPFS in Section 4.4.1, Section 4.4.2 and Section 4.4.3

4.4.1 The `virtio-fs` driver in the Linux kernel

We currently identify three performance-impacting implementation limitations in the `virtio-fs` driver in the Linux kernel: (1) no multi-queue support, (2) excessive usage of memory pages, and (3) poor handling of a full queue. This section will detail these three limitations of the `virtio-fs` driver.

The current implementation of `virtio-fs` in the kernel does not support multi-queue (128), 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 4.2a and Figure 4.2b.

Furthermore, 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 (128). 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).

When a new file system operation is issued to the `virtio-fs` driver and the (single) `virtio` queue is currently full, the current driver implementation schedules a retry of the given operation in 1 millisecond. The retry is scheduled and performed without any regard to ordering in respect to other new file system operations, so an operation can be live locked (i.e. indefinitely blocked because of the high load) from being put onto the `virtio` queue. We identify that this is the cause of the large standard deviation in throughput and latency with high I/O depths seen in Figure 4.2a and Figure 4.2b.

4.4.2 The DPU hardware

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) (111) poses a significant bottleneck. This bottleneck can be seen in the throughput difference of `DPFS-Null` (Figure 4.2a) and `DPFS-NFS` (Figure 4.2b). Where `DPFS-NFS` is only able to achieve $\sim 20\%$ 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.

As the DPU's hardware poses a processing bottleneck, it could turn out beneficial to adopt a passthrough architecture (like proposed in Section 3.2.2) that minimizes the amount of processing needed on the DPU. This architecture can be implemented in DPFS by means of a file system backend that sends the file system operations to a remote server instead of processing them locally.

4.4.3 Deploying DPFS in a cloud datacenter•

Currently, the DPFS framework can only expose the DPU as a single `virtio-fs` device to the host operating system. In data center deployments where only a single file system mount point is required, like containerized and bare metal deployments, this is not an issue. However, in virtual machine deployments where each virtual machine needs its own file system (i.e. `virtio-fs` device), like those common in the cloud, this limits DPFS to a single tenant. To support the wide variety of datacenter deployment types found in a cloud environment, DPFS needs to provide multiple `virtio-fs` devices to the host.

Furthermore, in this work, we implement the file system backends (i.e. `DPFS-NFS` and `DPFS-KV`) in userspace using the `DPFS-FUSE` API to decrease the number of context switches to the kernel and improve performance. However, this poses a hurdle for integrating DPFS in real-world data center deployments, as they employ many different file systems. The first 100 deployments in the 2023 edition of the IO500 list use 37 unique file systems (48). If all these 100 deployments would desire to move towards DPU-powered file system offloading with DPFS, it would require 37 file systems to be ported to `DPFS-FUSE`. It could, therefore, be beneficial to explore the integration of file system clients that reside in the Linux kernel (like is common with distributed file systems). This design allows DPFS to be plug-and-play (i.e. no code changes required) with any existing kernel-based file system and, thus, existing cloud deployments.

4. DPFS - A DPU-POWERED FILE SYSTEM VIRTUALIZATION FRAMEWORK

5

Evolving the DPFS framework into DPFS++

The DPFS framework and paper present a novel and promising approach to file system design by implementing a full-offload design using the NVIDIA BlueField-2. This work is extended in Chapter 4 by adding missing experiments and diving deeper into the limitations. This presents several opportunities for improving performance and adjusting to a cloud data center deployment environment. In this chapter, we act on these opportunities by implementing them in the evolved version of the DPFS framework called DPFS++.

The following extensions are designed and implemented in DPFS++ and detailed in this chapter:

- Two patches for the `virtio-fs` Linux kernel driver, that (1) improves request latencies during heavy workloads and (2) implements the previously missing multi-queue functionality, allowing for single tenant parallelism on the DPU (Section 5.1).
- A DPFS++ backend that exposes a file system mounted in the Linux kernel of the DPU, the plumbing is performed using the state-of-the-art `io_uring` interface (Section 5.2.
- A gateway passthrough implementation in DPFS++ that reduces the complexity of the data path on the DPU and introduces a gateway server into the system that resolves file system requests and is fully under the datacenter operator's control (Section 5.3.
- Multi-tenancy support that allows a single DPU to expose many `virtio-fs` devices and efficiently spreads the load of many tenants across the DPU cores (Section 5.4).

5.1 `virtio-fs` driver improvements

The research and engineering effort that has gone into DPFS has shed light on the current state and deficiencies of the `virtio-fs` driver in the Linux kernel. These deficiencies were laid out in Section 4.4.1, and we address two of these issues with Linux kernel patches. The first patch aims to reduce the latencies under heavy workloads (Section 5.1.1), and the second patch implements multi-queue functionality that aims to increase the available parallelism on the host and DPU (Section 5.1.2).

The two patches are based on the Linux kernel *6.5-rc1* release and can be found in the GitHub repository of the DPFS project [1]. The latency patch is currently under review in its fourth iteration (after back and forths with the maintainers) on the Linux Virtualization kernel mailing list [2]. The multi-queue patch is currently stalled as it cannot be properly tested due to an issue in the firmware of our NVIDIA BlueField-2.

5.1.1 Improving latencies under heavy workloads

A storage system's raw performance is only worth as much as the consistency at which it can deliver this performance. The issue of high tail latencies and high latencies under certain workloads is therefore considered an issue worth resolving.

In its current form, the `virtio-fs` driver in the Linux kernel retries a file system operation after 1 ms if it finds the `virtio` queue to be full. We regard this delay as large, considering that DPFS provides latencies an order of magnitude lower ($\sim$ 90 μsec using `DPFS-NFS`). There is, therefore, the need for a more performant approach to retrying operations.

A small patch for alleviating the latency issues in the `virtio-fs` driver

As a first effort towards alleviating the latency issue in the `virtio-fs` driver, we implemented a small patch for the driver (32 lines changed). With this patch, the driver does not retry queuing the request after a given timeout; the driver executes the two following steps:

- If the queue is full when a new request comes in, place the request on a dynamically allocated linked list inside the driver.

[1] `https://github.com/IBM/DPFS/tree/master/linux_patches`
[2] `https://lists.linuxfoundation.org/pipermail/virtualization/2023-July/067450.html`

- When other requests complete, take these completed requests off the queue. Immediately after removing the completed requests, attempt to fill the queue again with earlier requests from the linked list.
- On device *remove* (i.e., the file system is unmounted), the linked list is flushed to the device.

By immediately filling up the queue again with old requests before newer requests can be queued, requests cannot jump ahead anymore. Leading to the latencies being more predictable. Additionally, the hard-coded timeout value is removed as the retrying rate is directly tied to the speed at which the `virtio-fs` device operates.

5.1.2 Multi-threading a single tenant using multiple queues

For a single tenant, DPFS is only able to utilize a single DPU core for polling on the single `virtio` queue and resolving them to remote file system requests (e.g., NFS or RAMCloud). The NVIDIA BlueField-2 has eight cores (111), and newer DPUs like the NVIDIA BlueField-3 already double this with 16 cores (112). This leaves a lot of resources on the table for deployments with few tenants, like bare-metal servers with only a single tenant on the host CPU.

A possible approach for introducing multi-threading into DPFS would be to poll the single `virtio` queue on a thread and distribute the incoming requests onto N other threads. This work-dealing approach to multi-threading is used in the userspace FUSE library (3) as FUSE itself also only has one queue (i.e., the FUSE character device */dev/fuse*). This approach significantly increases performance under load compared to a single-threaded approach, however, the single queue and work dealing thread becomes a bottleneck under load (151). A bottleneck that increases in size alongside our core count. For this reason, a multi-queue approach has recently been pushed for FUSE (59). Many other high-performance systems efforts have also been toward multi-queue, notably block storage with NAND-flash devices (18) and networking with RDMA (70).

With these findings and the current trend of increasing core counts on DPUs in mind, it becomes apparent that multi-queue is the way forward for high performance on modern hardware. The `virtio-fs` protocol specification supports multi-queue (147). The multi-queue feature of the `virtio-fs` protocol is visually shown in Figure 5.1. The `virtio-fs` device has multiple queues and the capability to process the requests from these queues in parallel. Every CPU core on the host CPU, is assigned a `virtio` queue, and every request generated on core X is sent to queue X.

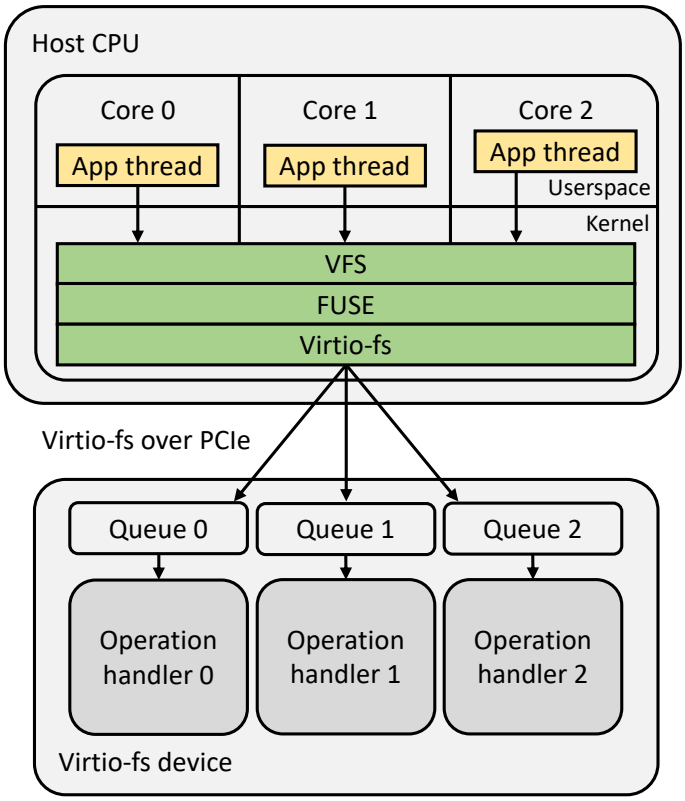


Figure 5.1: Diagram of the added multi-queue support in the `virtio-fs` Linux kernel device driver.

However, the `virtio-fs` driver implementation in the Linux kernel does not support multi-queue. It is noted in the source code as *"TODO multiqueue"* (128). To add multi-threading for a single `virtio-fs` device (i.e., a single tenant), a patch is implemented that adds the multi-queue functionality to the `virtio-fs` driver in the Linux kernel, and the DPFS++ framework is adapted to support polling on multiple `virtio` queues per `virtio-fs` device.

This patch and implementation were not completed because of issues with the NVIDIA BlueField-2 that occurred when using multiple `virtio` queues in combination with `virtio-fs`. When testing the feature, the DPU library would find an inconsistent state provided by the DPU's firmware. This issue was discussed with engineers from the storage virtualization team at NVIDIA. However, they cannot provide us with support or investigate the issue, as it is not a publically supported feature. But the next section will discuss the implementation details of the patch.

A patch for adding multi-queue support in the `virtio-fs` driver

To implement the multi-queue functionality into the driver, we draw inspiration from the `virtio-net` driver that also uses the same underlying `virtio` infrastructure of the Linux kernel and implements multi-queue support. The `virtio-net` driver maximizes core locality by creating a 1:1 mapping of cores and queues. The `virtio-net` driver with our multi-queue patch performs the following procedures related to multi-queue:

- On device *probe* (i.e., when the device is registered in the driver), create as many queues as there are cores.
- On device *probe*, the interrupt affinity of each `virtio` queue is set to its corresponding core.
- On device *probe*, the core-to-queue affinity of the networking layer is set up. This ensures that when the networking layer queues a new packet to the device, it can select the right queue depending on the current core.
- On device *probe*, callbacks are registered in the CPU hotplug subsystem that trigger when cores come online or go down (cores are dynamically turned on and off for energy saving). The callbacks renew queue affinities for interrupts and renew core-to-queue affinities in the networking layer. This can result in multiple queues being mapped to a single core when not all cores are online.
- On device *remove*, the queues are flushed, and affinities are cleaned up.

These procedures are trivially recreated in the `virtio-fs` driver, except for setting up the core to queue affinity for new requests. This is less trivial because, for `virtio-fs`, there is no higher layer (like the networking layer in `virtio-net`) that owns this responsibility. The patch solves this by keeping track of a multi-queue mapping inside the `virtio-fs` driver that maps core IDs to queue indices using an array, which is updated when a core comes online or goes down. When a new request needs to be sent to the device, the current core ID is used to index into the array and select the right queue.

5.2 Supporting kernel-based distributed file systems

A key issue that a data center operator currently faces when adopting DPFS is the issue of file system backend support. DPFS only supports NFS remote servers through the `DPFS-NFS` backend and RAMCloud remote servers through the `DPFS-KV` backend. NFS is

commonly deployed in today's data centers to virtualization the distributed file system backend (see Section 2.4.2); these deployments could utilize DPFS by connecting it to their existing NFS infrastructure. This is, however, undesirable as the DPU is already a virtualization layer under the purview of the data center operator; thus, keeping the NFS virtualization layer only increases cost and performance overhead. To remove the obsolete NFS virtualization layer from these deployments, DPFS must be able to connect to the internal distributed file system. Connecting DPFS to a distributed file system requires porting the file system client to the `DPFS-FUSE` API and programming model, a large task given that it needs to be performed for each distributed file system individually, of which there are currently many in deployment (48).

To solve this remote distributed file system support issue, DPFS++ implements the `DPFS-Kernel` file system backend that translates the `virtio-fs` operations down to the kernel-based file system operations, exposing a DFS mount-point of the kernel's virtual file system to the tenant on the host.

5.2.1 Accessing a file system efficiently in the modern Linux kernel

Traditionally, accessing the file system on a POSIX-compliant operating system was performed fully synchronously. All operations, such as opening a file, reading a file, or renaming a file, block the current thread until the operation has been completed. To achieve parallelism with file system operations, a program must utilize many threads that each issue a blocking operation. Today's workloads are massively parallel, and operating systems must be able to provide fast storage access to not bottleneck the workload. So the burden on file system I/O intensive applications to manage these I/O blocking threads has become a growing burden, especially as context switches between threads and the kernel has become an increasing bottleneck in modern systems (10, 159). Thus, there has been a push toward asynchronous APIs that do not require creating and managing many threads.

One of these APIs is the `io_uring` API of the Linux kernel (introduced in 5.1, released on May 6th 2019), which provides a generic interface for submitting operations to the kernel and retrieving operation completions from the kernel (36). The `io_uring` API is currently regarded as the state-of-the-art for interfacing with the Linux kernel and is the best-performing file system API that the Linux kernel supports (45). For the context of file system operations, one of the advantages over previous asynchronous interfaces of the Linux kernel (e.g., *libaio*) is the fact that it not only supports read and write operations but also metadata operations like *open* and *getattr* (36). This prevents the often high latency (95, 108, 154, 163) metadata operations from blocking the performance crucial

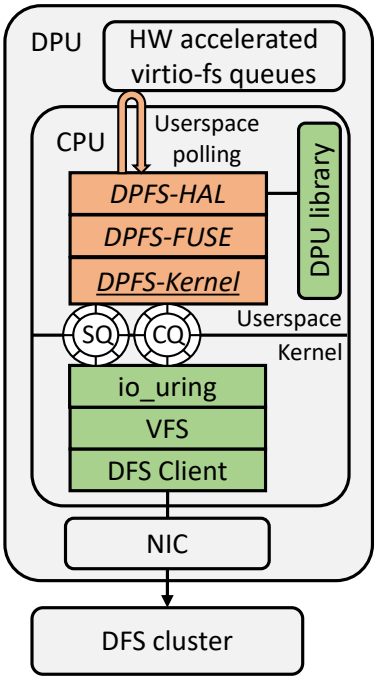


Figure 5.2: Diagram of the added multi-queue support in the `virtio-fs` Linux kernel device driver.

read and write operations in a single-threaded context (like that of a single tenant in DPFS++).

5.2.2 The design and implementation of the `DPFS-Kernel` backend in DPFS++

In Figure 5.2, the addition of the `DPFS-Kernel` backend is shown with regard to the other layers of the software stack residing on the DPU. The `DPFS-Kernel` backend implements the FUSE-like API that the `DPFS-FUSE` layer exposes and then translates each file system operation into a file system operation using either blocking POSIX function calls or asynchronous `io_uring` function calls.

Not all metadata operations can be translated using `io_uring` as many are not (yet) supported in `io_uring`. Table 1 in the appendix shows which operations are translated using blocking POSIX or asynchronous `io_uring` and which Linux kernel version is required for the asynchronous `io_uring` translation. The main takeaway from this table is that most of the file-based metadata operations (e.g., renaming or creating a symbolic link) are only fully supported from Linux kernel version 5.15 onward, and that most of the directory-based metadata operations (e.g., listing the files in a directory) are not supported in `io_uring`

as of this time. Since the programming model of `DPFS-FUSE` in DPFS++ is single-threaded per `virtio-fs` device, the file system operations that use blocking POSIX will block the complete `virtio-fs` device from the host perspective. In contrast, file system operations that are implemented using asynchronous `io_uring` operations are issued to the Linux kernel sequentially but executed in parallel by the Linux kernel and kernelspace file system client.

To reap completed file system operations off the `io_uring` completion queue, a second thread is spun up by `DPFS-Kernel` next to the `virtio-fs` device poller. The second completion reaper thread can operate in two modes: (1) **busy polling** on the completion queue (i.e., consuming a full core) or (2) **waiting** for new completions to arrive (i.e., blocks and suspends the thread until woken on a new completion). While busy polling will achieve higher performance, it will consume many resources, even during light workloads. Therefore, the choice of completion reaping mode relies on the deployment's performance requirements and available resources. In Section 6.3.1, the performance difference between busy polling and waiting is examined.

5.2.3 Advanced `io_uring` features

The `io_uring` API currently has three advanced features that aim to further increase the performance of communication between user and kernel space. These are (1) fixed buffers, (2) fixed files, and (3) kernel-side busy polling. With the fixed buffers feature, the application registers all the I/O buffers they will use in subsequent `io_uring` operations. By registering the buffers with the kernel, the kernel can pin and map these into kernelspace, reducing the number of data copies needed when issuing operations. With the fixed files feature, the application does not use POSIX file descriptors but uses opaque `io_uring` file descriptors that require less bookkeeping and optimization restrictions through not being POSIX compliant. With the kernel-side busy polling feature, the kernel will start up a kernel task that busy polls on the `io_uring` submission queue, removing the need for the application to use a system call to notify the kernel of a new operation submission.

The implementation of `DPFS-Kernel` in DPFS++ does not utilize these features for several reasons related to implementation complexity. The fixed buffers feature was not used as it requires registering the DPU library's underlying buffers for the `virtio-fs` requests. This task requires significant code changes to the proprietary NVIDIA SNAP source code in the case of the NVIDIA BlueField-2, as this DPU library has an internal code architecture (many layers of abstraction) that makes it difficult to extract these underlying buffers. The kernel-side busy polling feature was not utilized as it requires fixed buffers.

Without fixed buffers, the kernel cannot assume that the buffers will always be present and correctly mapped. Finally, the fixed files feature was not utilized as the opaque `io_uring` file descriptors cannot be passed to blocking POSIX file system function calls, which is a problem as `io_uring` currently does not (yet) support all file systems operations. A workaround for this issue would be the following approach: when the host opens a file on the `virtio-fs` device, open the file using both `io_uring` and POSIX and bookkeep both file descriptors for later use with both APIs. This is, however, not currently implemented in DPFS++, and thus, fixed files are not utilized.

5.3 Escaping the DPU with gateway passthrough

In Section 4.4, the bottlenecks of DPFS are laid out, a significant portion of which can be attributed to the inherently intensive file system processing (translating operations, bookkeeping inodes, network processing, etc.) performed on the NVIDIA BlueField-2's hardware. This shows the hardware limitations of the BlueField-2's hardware in effect with the DPFS framework. Given these hardware limitations, it becomes a logical step in the DPFS project to explore the passthrough design options proposed in Section 3.2.2 to escape the relatively slow DPU hardware as quickly as possible.

The BlueField-2 hardware is significantly underpowered compared to a modern data center-class server. To quantify this claim, we perform a small experiment using the Geekbench 6.1 benchmark (62) CPU benchmarking tool that compares the BlueField-2 against a modern AMD EPYC server (a Ceph node from Table 6.1). The results of this experiment can be found in Table 5.1. This experiment shows that a modern data center-class server contains 8.4× more compute power than the NVIDIA BlueField-2 DPU. Here, it is interesting to note that depending on the benchmark, the difference in performance can vary widely. When looking at benchmarks that are more FLOPs heavy (i.e., number crunching), like applying photo filters, instead of a more graph processing heavy workload like code compilation, the difference in performance increases from 4× to 22.6×.

To limit the scope of this research and engineering on DPFS++, we limited the design and implementation of gateway passthrough in DPFS++ to not support nor account for multi-tenancy.

5.3.1 The design of gateway passthrough in DPFS++

The gateway passthrough design detailed in Section 3.3.2 is, in short, when the DPU receives a file system operation from the host, the DPU will send, without processing the

Benchmark	NVIDIA BlueField-2 DPU	Modern AMD EPYC server	
Single-core suite	420 points	1648 points	3.9×
Multi-core suite	1969 points	16529 points	8.4×
Clang	547 points	2171 points	4×
Photo filter	4410 points	99715 points	22.6×

Table 5.1: Comparison of Geekbench 6.1 results on the NVIDIA BlueField-2 DPU compared to a modern AMD EPYC server with two benchmarks highlighted. Full results can be found at `https://browser.geekbench.com/v6/cpu/compare/1835864?baseline=1835749`.

operation, to a remote gateway that processes the operation and resolves the result using a remote distributed file system.

The gateway passthrough design requires two software components: a DPU client that sends the requests to the gateway and a gateway that receives them and resolves them to the remote distributed file system. These two components fit neatly into the architecture of DPFS, as both can build on top of the existing `DPFS-HAL` and `DPFS-FUSE` infrastructure that DPFS already has. In Figure 5.3, this design is shown with respect to the other layers of the DPFS software stack. The DPU client consumes the `DPFS-HAL` layer of the framework to use the DPU hardware and gathers `virtio-fs` requests to send to the gateway without doing any of the requests processing. The gateway implements the `DPFS-HAL` interface, where instead of using a DPU library to gather `virtio-fs` requests, it receives the requests from the DPU client. By implementing the `DPFS-HAL` interface, the existing `DPFS-FUSE` layer and its backends are able to run on the gateway without requiring code changes.

5.3.2 Implementation of gateway passthrough in DPFS++

One of the key advantages of the DPU-powered gateway passthrough design over the NFS gateway design is that the data center operator has full control over the networking stack through the abstraction that the DPU and `virtio-fs` protocol provide. Therefore, we can create a high-performance implementation that does not rely on previous standardization efforts.

The key consideration in implementing the gateway passthrough in DPFS++ lies in the communication between the DPU and gateway components. The communication pattern putting `virtio-fs` over the network can be characterized as RPC, as each file system request is, in essence, a remote procedure call (with a request and response). Our DPU (the NVIDIA BlueField-2) supports 100Gbs InfiniBand RDMA that can be accessed from

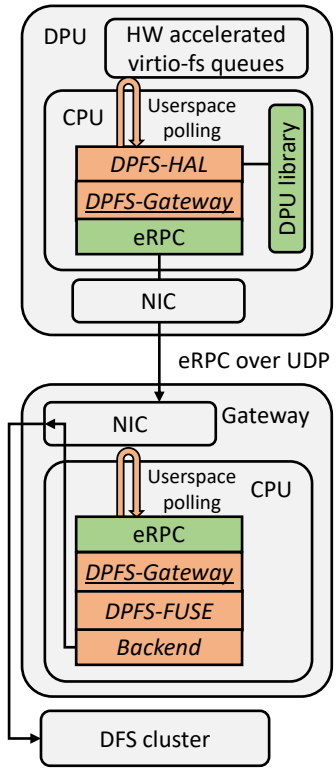


Figure 5.3: Diagram of the gateway passthrough design and implementation in DPFS++.

userspace on the DPU. RDMA is regarded as the state-of-the-art method for transferring data across the network. However, it faces several scalability and implementation issues (15, 49, 78). We, therefore, choose to implement the gateway passthrough using the state-of-the-art userspace RPC library *eRPC* that works on top of lossy Ethernet (69).

Figure 5.3 shows how the DPU and gateway components use the eRPC library to send the `virtio-fs` requests and responses over the wire. The DPU component can operate in either single-threaded or two-threaded polling modes. The single-threaded polling mode alternates between `virtio-fs` and eRPC polling. `virtio-fs` polling is performed using `DPFS-HAL`, and in case a `virtio-fs` request was successfully received by the DPU, it is translated into the wire protocol and placed on a queue for eRPC to handle. The eRPC polling runs the eRPC *event loop*, which sends queued `virtio-fs` request packets and polls on the network interface for `virtio-fs` completion from the gateway. The alternation is 1:1, so for every `virtio-fs` device poll, an eRPC event loop is run. The second mode, two-threaded polling, uses one thread to poll the `virtio-fs` device using `DPFS-HAL` and one thread for running eRPC event loops in parallel. By assigning a thread to each of the two tasks, the system's throughput is expected to increase. Still, the communication

between the two threads (which eRPC handles via locking the networking queues) might hamper performance. This hypothesis could not be tested due to time constraints in our evaluation suite (Chapter 6).

5.3.3 Layout of the wire-protocol

To keep the complexity and scope of the implementation in check, the wire protocol foregoes advanced features found in standardized remote storage and file system protocols, like automatic fail-over (156) or load balancing (109). The goal of the protocol is to be closely coupled to the request format of `DPFS-HAL` to minimize the complexity of the data path (i.e., minimize the number of data copies and transformations). When a `virtio-fs` request is received by `DPFS-HAL`, it is handed to the consuming layer (in this case, the DPU component of `DPFS-Gateway`) via the callback handler shown in Listing 1.

```
typedef int (*dpfs_hal_handler_t) (void *user_data,
                                   struct iovec *fuse_in_iov, int in_iovcnt,
                                   struct iovec *fuse_out_iov, int out_iovcnt,
                                   void *completion_context, uint16_t device_id);
```

Listing 1: Function signature of the callback handler exposed by `DPFS-HAL` in DPFS++. The device_id argument is introduced by the multi-tenancy support of DPFS++.

The relevant arguments for constructing the wire protocol of RVFS are second to fourth. These four variables contain the input and output buffers that translate to `virtio` descriptors in the DPU library. When sending a new incoming `virtio-fs` request to the gateway, the input buffers need to be sent to the gateway by the DPU, and inversely, when sending a `virtio-fs` response from the gateway to the DPU, the output buffers need to be sent from the gateway to the DPU. The gateway does need to be explicitly told about the output buffers, as this determines how much data can be read in a file-read operation. With these considerations in mind, a minimal and direct translation wire format is constructed. Table 5.2 and Table 5.3 show the wire formats for the RVFS protocol sent over eRPC in DPFS++ and describe the contents of the eRPC request and the eRPC reply.

5.4 Multi-tenancy in a dynamic cloud environment

A fundamental property of a cloud data center is the sharing of resources between multiple tenants. The sharing of resources is not set up in a static configuration; the cloud data

Bytes	Data type	Name	Description
4	int32	num_input_descs	The number of input descriptors in this request
8	uint64	input_desc_len	The number of bytes in the descriptor following this uint64
...	raw data	input_desc_data	Input descriptor data bytes
4	int32	num_output_descs	The number of output descriptors available for the reply
8	uint64	output_desc_len	The length of an output descriptor

Table 5.2: The wire format of `DPFS-Gateway` requests in DPFS++. num_output_descs indicates how many instances of desc_len and desc_data are contained in the request, and similarly for num_input_descs and output_desc_len.

Bytes	Data type	Name	Description
...	raw data	desc_data	Descriptor data bytes

Table 5.3: The wire format of `DPFS-Gateway` responses in DPFS++. The number of descriptors and their lengths are not in the response, as the DPU component stores this information when sending out the request.

center is a dynamic environment with tenants requesting and deallocating resources at any time. To support cloud data center deployments, multi-tenancy support is crucial to DPFS++. This section will explain the multi-tenancy facilities of the `virtio-fs` protocol and a design for the polling scheduler in DPFS++.

5.4.1 Design of multi-tenancy support in DPFS++

Multi-tenancy requires two major design considerations: how the DPU exposes itself as multiple file systems to the host CPU and how the DPU schedules and performs the work required for these file systems. This subsection will detail the DPFS++ design for multi-tenancy support.

Connecting a single DPU to multiple tenants

The DPFS framework uses the `virtio-fs` over PCIe protocol to expose itself as a file system to the host. The PCIe interconnect standard has two solutions for dynamically adding and removing devices from the host CPU, namely PCIe *hot-plug* and *Single Root I/O Virtualization* (*SR-IOV*). PCIe hot-plug needs to be explicitly supported by the kernel and the device driver, and it is currently not supported in the Linux `virtio-fs` device driver. SR-IOV allows a single root (i.e., a single device) to virtualize itself into multiple virtual devices through the SR-IOV extension to the PCIe standard (123). The SR-IOV

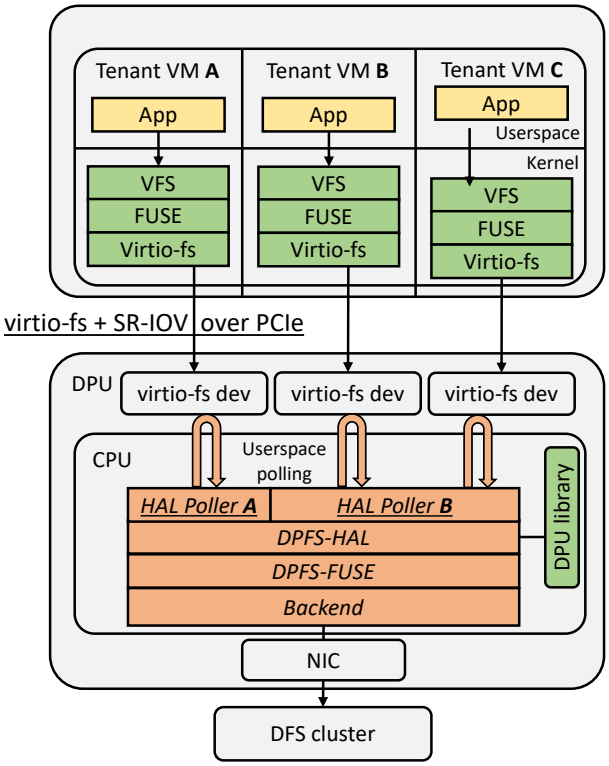


Figure 5.4: Diagram of the design and implementation of multi-tenancy support in DPFS++.

extension allows a PCIe device to expose physical functions and virtual functions, where many virtual functions are associated with a single physical function. A physical function is presented at the *reset* (i.e., boot) of the device and allows the host to manage the device and create a certain number of virtual functions during runtime. These virtual functions can then be passed through to a virtual machine to allow the virtual machine to consume the device. This allows the device to virtualize and share its resources between the virtual functions internally. It removes the need for the hypervisor (the virtualization layer on the host) to virtualize the device through software, thus efficiently virtualizing its resources with hardware assistance in a multi-tenancy environment.

The SR-IOV extension is explicitly supported in the `virtio` standard (147), is supported in the Linux kernel, and is supported in commercially available DPUs like the NVIDIA BlueField-2 (26, 97, 111). Thus, it is the (only) way to offer dynamic multi-tenancy support for `virtio-fs` in DPFS++.

Challenges for adding multi-tenancy support to DPFS, and learning from SPDK

In DPFS, a single core is consumed on the DPU to poll on the `virtio` queue for `virtio-fs` requests. When extending this model to support multiple `virtio-fs` devices, we find the following challenges:

1. During low-load scenarios, the core is excessively consumed without doing meaningful work.

2. When extending this design to multiple `virtio-fs` devices, N devices would consume N cores. This is problematic when scaling to many tenants when the current generation of commercial DPUs generally have 16 cores (26, 97, 111), especially when considering that the DPU cores are commonly also tasked with other workloads like datacenter orchestration (12, 113, 125, 164) and other custom offloads (91).

The problem of efficient userspace storage device polling with multi-tenancy is not new, as other frameworks exist that aim to solve the problem. Most notably, the Storage Performance Development Kit (SPDK) framework (159) allows users to consume a storage NVMe device entirely from userspace through polling. However, this is, in essence, the reverse point of view from DPFS, where DPFS polls to expose itself as the actual storage device (i.e., acting as a device implementation), SPDK polls to consume the storage device (i.e., working as a device driver). To poll multiple storage devices efficiently, even during low-load scenarios, SPDK contains a dynamic scheduler designed to increase power efficiency and reduce CPU utilization (63).

The dynamic scheduler of SPDK centers around the idea of *busy ticks* and *idle ticks*, which allows the scheduler to calculate the *busy time* of a poller (i.e., what percentage of time a poller was handling requests). When a poller's busy time drops below a configurable rate, then it is marked as *idle* and is moved off the set of cores (called *reactors*) that handle *busy* pollers and onto a single core that is designated for idle pollers. A reactor suspends itself when it has no pollers assigned anymore. When an idle poller's busy time rises above that configurable percentage again, it is moved back to one of the reactors to poll. By dynamically spreading the busy pollers across a set of cores and possibly suspending cores when there is a low load on the system, it increases the energy efficiency without sacrificing performance and solves the same problems that DPFS faces with multi-tenancy.

5.4.2 Implementation of multi-tenancy support in DPFS++

For the implementation of multi-tenancy support, we opt to use a simpler *static* version of the dynamic scheduler design; it is left up to future work to implement and evaluate the complete scheduler design. The scheduler implemented in the system statically distributes the N devices evenly across the P cores. This design and its implementation are visually represented in Figure 5.4. For instance, consider a scenario where DPFS++ is configured with three devices and two cores, like in Figure 5.4. In such a case, each core will be assigned a poller thread: poller A will perform busy polling on one device, and poller B will perform busy polling on two devices. As a result, each device will receive 50It's worth noting that just like in DPFS, a single tenant (which is represented by a `virtio-fs` device) can only be served by a single DPU core at a time if multi-queue is not enabled in the `virtio-fs` driver, which is currently not enabled (as stated in Section 5.1.2). Therefore, although DPFS++ is designed to be multi-threaded with its multi-tenancy feature and dynamic scheduler, this can only be achieved through multiple `virtio-fs` devices.

SR-IOV support on the BlueField-2

The NVIDIA BlueField-2 DPU supports the SR-IOV (111), and there is documentation on how to employ it with the `virtio-net` network virtualization stack that NVIDIA offers (110). However, since `virtio-fs` is not a publically supported feature, there is no documentation on using SR-IOV with `virtio-fs`. When following the `virtio-net` firmware configuration instructions and replicating them on the `virtio-fs` firmware configuration, we find that the NVIDIA SNAP library returns an undocumented error from the firmware when trying to create `virtio-fs` devices. Similarly to the multi-queue issue discussed in Section 5.1.2, this issue was also addressed with engineers from the storage virtualization team at NVIDIA, and they were unable to help us, as it is not a publically supported feature.

For this thesis and research scope, we conclude that the BlueField-2 does not support SR-IOV for `virtio-fs` in the firmware.

Employing physical functions and *mock* physical functions for multi-tenancy

As the BlueField-2 does not support SR-IOV and is the only DPU we have access to, the multi-tenancy implementation of DPFS++ can only employ physical functions. When evaluating the performance of the multi-tenancy support, this limitation does not affect the results because in the `virtio-fs` protocol, physical and virtual functions have the

same functionality. The BlueField-2's firmware configuration allows us to configure up to 10 physical `virtio-fs` devices.

The only issue caused by not supporting virtual functions in our evaluation is that we cannot dynamically create and destroy `virtio-fs` devices and, therefore, have to configure the number of devices at boot. This is problematic as a reboot takes around ~ 5 minutes, and rebooting the system is hard to automate in an evaluation suite. Performing all performance evaluations with the full ten `virtio-fs` devices configured is problematic, as all devices would be polled regardless of whether they are used in the experiment, thus consuming excessive resources. To work around the excessive resource utilization in this scenario, the multi-tenancy implementation in DPFS++ supports so-called *mock* `virtio-fs` physical functions. The devices for these physical functions are created like non-mock devices. However, they are only polled once a second (1 IOPS). This allows the device to be correctly initialized on the DPU and host kernel without consuming extra resources. In the worst case, where ten devices are configured, and nine are mock devices, the DPU wastes 9 IOPS on these devices while a single DPU core can sustain upwards of 300k IOPS. The overhead can, therefore, be entirely disregarded.

The impact of multi-tenancy support on the file system backend

Multi-tenancy is a functionality that cannot be left transparent towards the file system backend, as with multiple tenants, new considerations come into play, such as quality of service and security. In the multi-tenancy support of DPFS++, these considerations are left entirely to the file system backend implementation. It is left for future work to add infrastructure to DPFS for providing a certain quality of service and security guarantees. This section will go into detail on how the multi-tenancy functionality is exposed to the file system backend and how the `DPFS-NFS` and `DPFS-Kernel` backends integrate this (the other backends do not support multi-tenancy in DPFS++).

With multi-tenancy, DPFS++ can be configured to run with N devices that are polled on P cores, effectively introducing multi-threading into the file system programming model of the backends. The requests are all passed to the file system backend using the function handler shown in Listing 1, with the `void *user_data` pointing to a singular data structure provided by the file system backend. The file system backend is, therefore, responsible for correctly handling the parallelism present in the system. To find out the parallel environment in which the backend is currently running, it can use the `uint16_t dpfs_fuse_nthreads(struct dpfs_fuse *fuse)` function for retrieving P (static value) and `uint16_t dpfs_hal_thread_id(void)` function for retrieving the thread ID (starting

at zero) of the current request handler context. The DPFS-NFS and DPFS-Kernel backends create P connections to their respective API (NFS over TCP connections, and io_uring rings) on startup. When the request handler is called, it uses the current thread ID to select a connection. To handle the request completions on these connections, the backends create threads for reaping the completions.

DPFS-NFS creates N threads that wait (blocking and suspending the thread after a small timeout) for completions on their connection, and DPFS-Kernel has a configurable amount of completion reaping threads and a configurable reaping method (see Section 5.2.2) This file system programming pattern allows a single file system instance to minimize the contention between the threads.

```
typedef void (*dpfs_hal_register_device_t) (void *user_data, uint16_t device_id);
typedef void (*dpfs_hal_unregister_device_t) (void *user_data, uint16_t device_id);
```

Listing 2: Function signatures of the device register and unregister callback handlers exposed by DPFS-HAL in DPFS++

Even though the current multi-tenancy implementation does not support dynamically bringing devices up and down, the API that DPFS-HAL exposes to the file system backends is created with dynamicity in mind. When a device is created or destroyed by DPFS-HAL, the register or unregister callbacks of the file system backend are called; these callbacks are shown in Listing 2. Using the void *user_data argument, the function can access the backend's state and register a device with the device identifier provided in uint16_t device_id. When the host issues a file system operation on this device, the backend can identify the request's origin during the device ID provided in the request handler callback (shown in Listing 1). This allows the file system backend to isolate the different devices (which can be assigned to other tenants on the host by the operator through the hypervisor) by, for example, providing them with their subtree of the remote file system. DPFS-NFS and DPFS-Kernel do not perform device isolation and offer a single remote file system tree to all configured devices.

Without Single Root I/O Virtualization (SR-IOV), it is not possible to dynamically bring devices up and down as PCIe hot-plug is not supported. The current implementation in DPFS++ initiates the device register and unregister callbacks on startup after the backend has been initialized but before polling any of the devices.

6

Evaluating the DPFS++ framework

More often than not, the art of computer systems lies in experimentation. A computer system contains many complex layers, spanning both software and hardware. The complexity of a system and its interactions only become apparent during experiments, whether running real-world or synthetic workloads. A thorough evaluation of a computer system is therefore crucial to furthering our field of research.

In this chapter, we evaluate the extensions presented in DPFS++ using multiple experiments that compromise synthetic and real-world workloads. Section 6.1 describes the experimental setup for reproducing the results presented reliably. Our evaluation contains the following four sets of experiments (each corresponding to a section in Chapter 5):

- In Section 6.2, we evaluate the `virtio-fs` driver latency patch, which was previously introduced in Section 5.1.1.

- In Section 6.3, we benchmark the `DPFS-Kernel` file system backend, designed and implemented in Section 5.2, using a RAM-based file system and a Ceph distributed file system client (154).

- In Section 6.4, we evaluate the `DPFS-Gateway` file system backend, detailed in Section 5.3, when using a gateway and guided passthrough design (these two designs are introduced in Section 3.2.2 and Section 3.3.2).

- In Section 6.5, we measure the impact of the multi-tenancy support in DPFS++, described in Section 5.4, when using up to 8 tenants.

6.1 Experimental setup and reproducibility

This section presents the details of our experimental setup to ensure reproducibility and give experimental confidence. Our setup consists of four hardware systems: (1) the host server, (2) the DPU attached to the host server, (3) the gateway server (NFS for RAM-backed experiments and RVFS), and (4) the Ceph cluster (also running an NFS gateway). These machines' hardware specifications and software versions are specified in Table 6.1. We use the following four workloads in the experiments: a synthetic file system I/O generator *fio* (14), two real-world key-value databases *Redis* (92) and *Rocksdb* (30), and a file system workload simulator *Filebench* (141).

	Host	**DPU NVIDIA BlueField-2**	**Gateway server**	**Ceph nodes (4x)**
CPU	2x Intel Xeon E5-2630 v3, 2.4GHz, 8 cores/socket	8x ARMv8 A72 64-bit cores at 2.75 GHz	2x Intel Xeon CPU E5-2690, 2.9GHz, 8 cores/socket	2x AMD EPYC 7453, 3.5GHz, 28 cores/-socket
Memory	128 GiB, DDR4 1,866 MT/s	16 GiB, DDR4 3200 MT/s (18 GiB/s)	128 GiB, DDR4 1,333 MT/s	256 GiB, DDR4 3200 MT/s
Network	100Gbps Ethernet via DPU, connected via PCIe 3.0 x16 (14.7 GiB/s)	Integrated Mellanox ConnectX-6 Dx 100Gbps Ethernet	Mellanox ConnectX-5 100Gbps Ethernet	Mellanox ConnectX-6 Dx 100Gbps Ethernet
Storage	n/a	n/a	n/a	3x Micron 7400 PRO 3D TLC, NVMe over PCIe 4.0, 3.5 TiB, head node has no storage
Software	Ubuntu 22.04.3 LTS, linux: 6.2.0, libcephfs2: 17.2.6-0ubuntu0.22.04.1, fio: v3.35, rocksdb: v8.3.2, redis: v7.0.12, filebench: commit 22620e60	BSP: 3.9.3, Ubuntu 20.04.6 LTS, linux: 6.2.16, Mellanox OFED Linux drivers: 23.04-1.1.3.0, libcephfs2: 15.2.17-0ubuntu0.20.04.4, liburing 2.3	Ubuntu 22.04.3 LTS, linux: 6.2.0, libcephfs2: 17.2.6-0ubuntu0.22.04.1	Ubuntu 22.04.3 LTS, linux: 5.15.0-83-generic, ceph: 17.2.6 (quincy)

Table 6.1: Hardware specifications and software versions of the machines (including the DPU) used in the evaluation of DPFS++.

The following list consists of several experimental details that we have found to influence the performance measurements:

- Workloads that use a single thread (i.e., fio) are pinned to the host CPU core where the DPU's interrupts are mapped.
- Multi-threaded workloads (i.e., filebench, redis, and rocksdb) are pinned to the host CPU NUMA node local to the DPU's PCIe connection.
- C-states are disabled during latency experiments on the host.
- C-states are always disabled on the gateway server and the Ceph nodes.
- Direct I/O is always used in fio workloads on the host and in `DPFS-Kernel` when accessing the locally mounted file system.
- File metadata caching is always turned on the host and DPU.
- The filebench workloads are default except for the I/O size; this is 128 KiB instead of 1 MiB. This change accommodates the lack of true >128 KiB I/O size support in DPFS++.
- Unless specified otherwise, `DPFS-Kernel` is used in polling mode.
- The NFS server on the gateway server is configured with: "async, no_subtree_check", and the NFS client parameters used on the host are: "async").
- In Section 6.3, Section 6.4 and Section 6.5 the host's Linux kernel is patched with the `virtio-fs` device driver latency patch from Section 5.1.1.

The NVIDIA BlueField-2 DPU used in our experimental setup runs the BlueField DPU BSP 3.9.3 software, but the Linux kernel is updated to v6.2.16 (custom compiled using the BlueField configuration), and Mellanox OFED is updated to 23.04-1.1.3.0 for the Linux 6.2 compatible drivers. By default, BSP 3.9.3 ships with Linux kernel v5.4, which, as discussed in Section 5.2.2, does not support all the latest file system operations in `io_uring`. Therefore, the kernel was updated to v6.2 (the latest kernel version supported by Mellanox OFED as of September 2023). To be able to access `virtio-fs` functionality on the NVIDIA BlueField-2 DPU, a prototype firmware is required that was generously provided by NVIDIA through a research collaboration with IBM Research.

All the code of DPFS++, the scripts that run the experiments, the results of said experiments, and instructions on setting up the experimental setup can be found at `https://github.com/IBM/DPFS`.

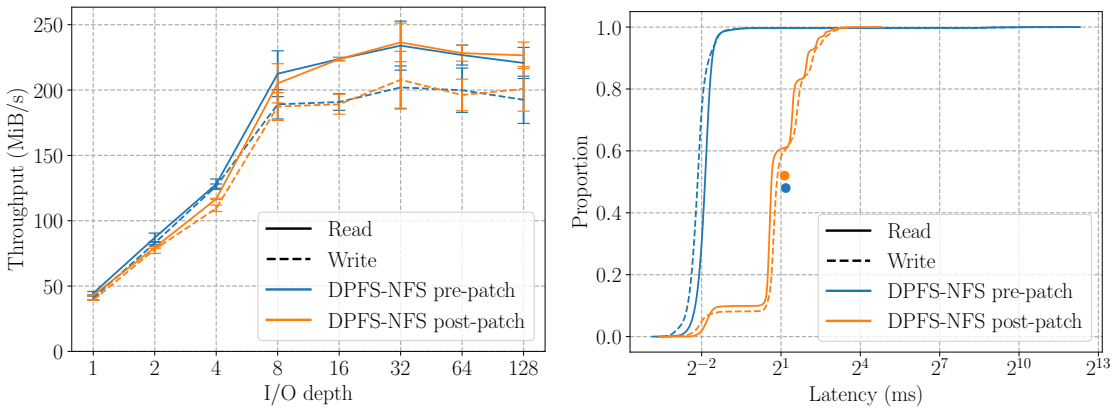
6.2 `virtio-fs` latency driver patch

We start this evaluation chapter by evaluating the `virtio-fs` patch detailed in Section 5.1.1 and is currently in the process of being upstreamed to the Linux kernel. The goal of this patch is to decrease the variation in operation latency that occurs when the system is under heavy load, specifically when the number of outstanding requests is larger than the depth of the `virtio-fs` device's queue. We use the `DPFS-NFS` file system backend with a `tmpfs` file system on the remote NFS server to evaluate this patch. Like the experiments in Section 4.3, the `tmpfs` file system allows us to eliminate the storage medium on the remote server as source of performance variability. The userspace NFS backend utilizes the NVIDIA XLIO TCP offloading library, which (because of currently unresolvable software issues) limits the queue depth of the `virtio-fs` device to just 64 descriptors, which translates to a queue depth of only 16 operations.

The evaluation of the latency driver consists of a single experiment using `DPFS-NFS`, where the gateway server (i.e., the NFS server) exposes a RAM-based `tmpfs` over the NFS protocol. The host performs random 4 KiB I/O (read and write) using fio at various I/O depths on the `virtio-fs` device. Figure 6.1a shows the throughput in MiB/s on the y-axis and the I/O depth on the x-axis of the `virtio-fs` driver in its pre- and post- patch forms. We find that for low I/O depths, the throughput with the latency patch is lower by less than 5 MiB/s, however, at higher I/O depths, we find no performance difference.

Figure 6.1b shows the results from this same experiment, but only at I/O depth 128, in a cumulative distribution function (CDF) graph. At the I/O depth of 128, there are 8× more outstanding operations than can fit in the `virtio-fs` queue. The y-axis represents the fraction of operations below the latency shown in milliseconds on the x-axis. The dots represent the average latency of the pre and post-patch driver. Using this projection of the experiment's results, we find that while almost all of the latencies observed with the patched driver are higher than without the patch because the order in which requests are put into the queue and resolved is now fair (i.e., no overtaking). The average latency is 3% and 5.2% lower with the patch for reads and writes, respectively, because the latency tail is significantly cut short. The 99.99th percentile latency is reduced from 1.8 seconds to 14 ms for reads, and from 2 seconds to 16.8 ms for writes.

Thus, we conclude that, while the patch does not significantly affect the overall throughput performance of `DPFS-NFS`, it does reduce the average latency by 3-5% and tail latencies by more than two orders of magnitude, thus achieving its goal of reducing the latency variability.



(a) Random 4 KiB I/O throughput of `DPFS-NFS` pre and post `virtio-fs` latency patch

(b) Random 4 KiB I/O with an I/O depth of 128, CDF of `DPFS-NFS` pre and post `virtio-fs` latency patch, the dots represent average latency

Figure 6.1: `virtio-fs` driver latency patch experiments

6.3 `DPFS-Kernel` with a distributed Ceph storage cluster

The `DPFS-Kernel` backend can mirror any file system mounted in the DPU's Linux kernel to the host CPU over a `virtio-fs` device. To evaluate this backend, we use two file systems: a RAM-backed `tmpfs` file system to evaluate the communication overhead introduced (Section 6.3.1 and a file system client for Ceph (Section 6.3.3 and Section 6.3.4). Ceph (154) is a widely used distributed storage system (48) that supports block, object, and file storage, and is an official Linux foundation project. For the purposes of this thesis and evaluation, it serves as a best effort of replicating a distributed file system that could be deployed in a cloud data center, at the small (in vitro) scale of our experimentation.

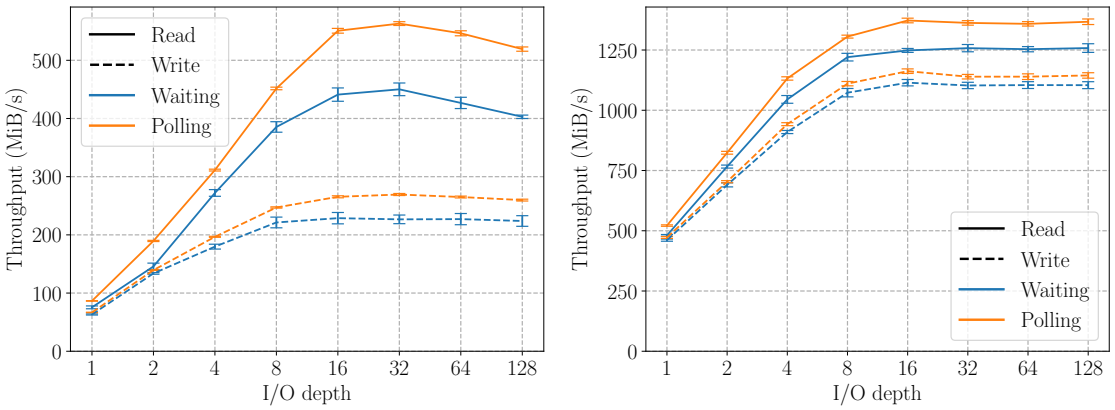
In these experiments, the `virtio` queue depth of DPFS++ is configured at 512 descriptors, which translates to roughly 128 operations.

6.3.1 Tmpfs - `DPFS-Kernel` virtualization overhead

To evaluate the kernel-based file system mirroring capabilities of `DPFS-Kernel` using `io_uring`, independently of the remote storage backend, we use the RAM-backed file system `tmpfs` that runs in the DPU's Linux kernel. We evaluate both the waiting and polling `io_uring` modes (Section 5.2.2). The waiting mode consumes less power but has to incur the overhead of returning from power-saving and context switching, generally resulting in lower performance than the polling mode.

Figure 6.2 shows the results of the random I/O throughput experiment performed using fio by the host CPU on the `virtio-fs` device that the DPU exposes. For random reads, we find that the difference between waiting and polling is around 116 MiB/s for the 4 KiB block size and 109 MiB/s for the 64 KiB block size, at an I/O depth of 128. For random writes, the difference is 36 MiB/s and 40 MiB/s for 4 KiB and 64 KiB, respectively, at I/O depth 128. The standard deviations of the throughput (shown using whiskers on every data point), show that the throughput does vary significantly more when using waiting than when polling. For 4 KiB reads, the standard deviation of the throughput when waiting is as large as 11.7 MiB/s (I/O depth of 16) compared to 4.2 MiB/s when polling.

Overall we report that `DPFS-Kernel` using `io_uring` with a RAM-backed file system can support a throughput upwards of 1 GiB/s for the larger block size of 64 KiB, 133k random 4 KiB read IOPS and 66k random 4 KiB write IOPS. Overall, the polling mode of `io_uring` increases throughput by 18% on average.



(a) Random 4 KiB I/O throughput of `DPFS-Kernel` with `tmpfs`

(b) Random 64 KiB I/O throughput of `DPFS-Kernel` with `tmpfs`

Figure 6.2: Virtualization overhead experiments of `DPFS-Kernel`, comparing the waiting and polling completion methods of `io_uring`

6.3.2 The Ceph Cluster

The Ceph distributed storage system has many configuration options which makes finding the optimal configuration challenging. Also, is not feasible to replicate the performance characteristics of a large-scale cloud deployment using a small deployment (like our experimental setup). We, therefore, choose to create a simple deployment based on the default Ceph configuration. The Ceph cluster comprises four modern AMD EPYC servers, whose specifications are shown in Table 6.1. Three of these servers are storage nodes, each con-

sisting of three enterprise Micron NVMe drives configured with a single *OSD* in Ceph. In Ceph, an OSD (Object Storage Daemon) is a process that manages the drive and handles reads and writes to that drive. The fourth server does not contain any storage drives used in the cluster. It acts as the head node of the cluster, running the CephFS (file system) server, the NFS gateway server, manages the file system metadata, and runs other management-related processes.

To connect to the remote Ceph cluster, we use the NFS client on the host (as our baseline for a cloud-like Ceph deployment) or `virtio-fs` on the host with `DPFS-Kernel` on the DPU. The `DPFS-Kernel` backend is configured to mirror a CephFS mount point (in the Linux kernel) to the tenant(s) on the host.

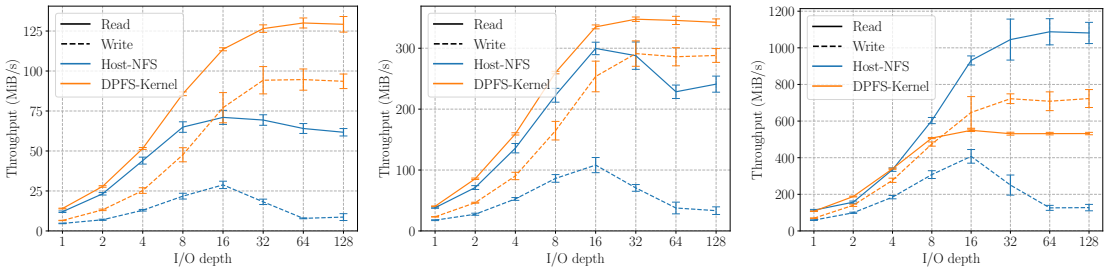
6.3.3 Ceph - Synthetic workloads

In Figure 6.3, the `DPFS-Kernel` with Ceph synthetic workload experiment results are shown for the 4 KiB, 16 KiB, and 64 KiB block sizes, comparing to the host NFS setup (connected to the NFS gateway exposed by the Ceph cluster). Similarly to the findings in the `DPFS-NFS` evaluation of Section 4.3.2, we find that, for the smaller block sizes of 4 KiB and 16 KiB in Figure 6.3a and Figure 6.3b, `DPFS-Kernel` achieves significantly higher throughput than host NFS at deep I/O depths. For random 4 KiB I/O at 128 I/O depth, the `DPFS-Kernel` throughput is 2.1× and 10.8× higher for reads and writes, respectively. However, this is caused by a sudden drop in throughput at I/O depth 16 of up to 70% for 16 KiB writes when using the host NFS setup.

The experiment with a larger block size of 64 KiB in Figure 6.3c shows that, even with a large block size, `DPFS-Kernel` provides significantly higher write throughput than host NFS, 2.4× at I/O depth 16 and 8.5× at I/O depth 128. For 64 KiB random reads, however, host NFS does not exhibit the drop in performance when the I/O depth exceeds 16 as it did for 4 KiB and 16 KiB random reads. At the same time, `DPFS-Kernel`'s throughput starts flatlining at I/O depth 8, resulting in host NFS providing 2× higher throughput than `DPFS-Kernel` when running a 64 KiB random reads workload.

6.3.4 Ceph - Real-world workloads

Figure 6.4 shows the real-world experiments conducted on `DPFS-Kernel` connected to the remote Ceph storage, compared to the traditional cloud host NFS setup used in earlier experiments. `DPFS-Kernel` is used in both polling and waiting mode to investigate the



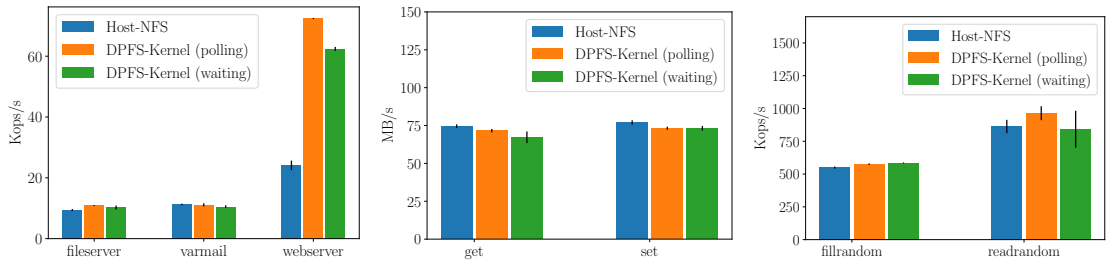
(a) Random 4 KiB I/O **(b)** Random 16 KiB I/O **(c)** Random 64 KiB I/O

Figure 6.3: Synthetic throughput (fio) experiments of `DPFS-Kernel` with Ceph

effect of the `io_uring` mode for real-world and file metadata workloads. The standard deviation in these graphs is the variance from run to run.

Comparing the host NFS setup and `DPFS-Kernel`, we find that their performance is comparable irrespective of the workload, except for Filebench webserver. The Filebench webserver workload performs 3× better using `DPFS-Kernel` with polling mode compared to host NFS. The Filebench webserver workload repeatedly performs three actions on random files: open, read the whole file, and close. The previous experiment of Figure 6.3c showed that for large block size reads (64 KiB in that experiment, Filebench uses 128 KiB), host NFS provides 2× higher throughput than `DPFS-Kernel` (also polling), opposite of what we find in this experiment. Furthermore, the two other Filebench workloads (fileserver and varmail) also open and close files at the same rate. We are therefore currently unable to explain the difference in performance between host NFS and `DPFS-Kernel` for the Filebench webserver workload.

Summarizing the overall impact of using polling versus waiting in `DPFS-Kernel` for the real-world workloads, we see that when the difference is significant, polling is always faster than waiting (6.8% on average overall), and polling has a lower standard deviation than waiting. However, the difference is negligible for most workloads (i.e., Filebench fileserver, Filebench varmail, Redis set, and Rocksdb fillrandom). The Filebench webserver and Rocksdb readrandom stand out as they show significantly significant differences in performance between the two `io_uring` modes. These two workloads are read-heavy, so it reaffirms the finding of Section 6.3.1 that the polling mode impacts the read I/O operations more than write I/O operations.



(a) Filebench **(b)** Redis **(c)** Rocksdb

Figure 6.4: Real-world workload experiments of `DPFS-Kernel` with Ceph (in polling and waiting mode) compared to host-NFS with Ceph

6.4 The gateway implementation of `DPFS-Gateway`

The gateway implementation of `DPFS-Gateway` introduces a new system design into the DPFS++ framework. It incorporates a gateway server that does all the processing and resolving of the file system operations; the DPU only functions as a naïve passthrough device of the `virtio-fs` requests. Given the hardware resource limitations of the Nvidia BlueField-2 DPU used in our experimental setup, we hypothesize that resolving the operations on modern server-class hardware (i.e., the gateway) will improve throughput compared to the full-offload deployment type of DPFS++ (e.g., running `DPFS-Kernel` on the DPU).

In this section, we perform synthetic and real-world workload experiments on `DPFS-Gateway` and compare it against the host NFS and `DPFS-Kernel` configurations. First, in Section 6.4.1, we run the `DPFS-Null` file system backend on the gateway to evaluate the maximum throughput that our gateway can provide. Last, in Section 6.4.2, we run the `DPFS-Kernel` file system backend on the gateway connected to our Ceph cluster (the cluster is described in Section 6.3.2).

In these experiments, `DPFS-Gateway` is always running in single-threaded mode (see Section 5.3), and the `virtio` queue depth of DPFS++ is configured at 512 descriptors, which translates to roughly 128 operations.

6.4.1 Finding the upper limits of `DPFS-Gateway`

By running `DPFS-Null` on the gateway, we can determine the amount of bandwidth available in the system, from the host CPU issuing file system operations to the `virtio-fs` device to the `DPFS-Gateway` and back. These results indicate the maximum performance of any file system backend when deployed `DPFS-Gateway`.

Figure 6.5 shows the results of this synthetic experiment. For the larger block size of 64 KiB, the throughput of `DPFS-Gateway` tops out at ∼1 GiB/s with an I/O depth of 8. At I/O depths deeper than 8 with the 16 KiB and 64 KiB block sizes, the throughput of reads and writes converge at around 800 MiB/s. For 4 KiB/s random I/O, we find that `DPFS-Gateway` can provide 114k read IOPS and 122k write IOPS.

These throughput numbers are 5× to 6× lower than the throughput numbers reported in Section 4.3.1, where `DPFS-Null` can achieve up to 5.87 GiB/s for reads and 4.32 GiB/s for writes (at I/O depth 16-32) when running on the DPU (i.e., measuring the bandwidth between the host CPU and the DPU). The 4 KiB IOPS that `DPFS-Gateway` provides is 2.9× and 2× lower than that of `DPFS-Null` on the DPU for reads and writes, respectively.

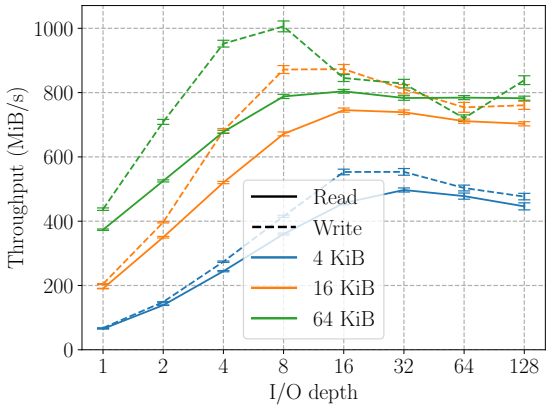


Figure 6.5: Synthetic random I/O workload (fio) of `DPFS-Gateway` with the `DPFS-Null` backend and one thread on the DPU, throughput for block sizes 4 KiB, 16 KiB and 64 KiB.

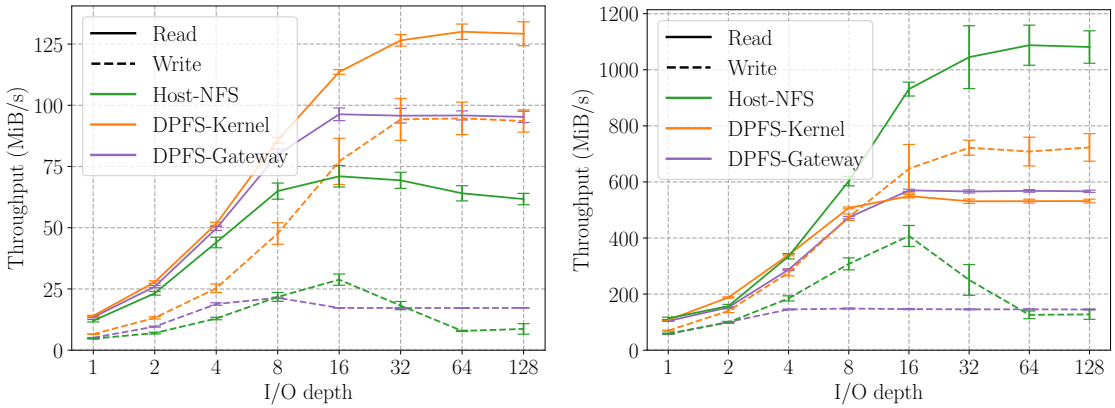
6.4.2 Evaluating gateway passthrough in DPFS++ using Ceph

The second set of experiments we run to evaluate the performance of `DPFS-Gateway` is `DPFS-Gateway` with the `DPFS-Kernel` backend connected (via the CephFS client in the Linux kernel) to our Ceph cluster. In this section, we refer to the `DPFS-Gateway` with `DPFS-Kernel` setup as simply `DPFS-Gateway`. We perform random I/O using fio with a block size of 4 KiB and 64 KiB. The results of these experiments are shown in Figure 6.6.

The two graphs show that writes `DPFS-Gateway` performs 5.5× and 3.9× worse than reads for 4 KiB and 64 KiB block sizes, respectively. This performance pattern does not appear when `DPFS-Kernel` runs on the DPU. Doing performance analysis using *perf* on the gateway server of `DPFS-Gateway`, we find that 2/3 of the wall clock time is being spent by copying `virtio-fs` request data around between buffers due to `DPFS-Gateway` and eRPC

requiring several data copies on the write file operation path. The read throughput of `DPFS-Gateway` is 30% lower than that of `DPFS-Kernel` for the 4 KiB block size and 6% higher for the 64 KiB block size.

Overall, we conclude that `DPFS-Gateway`, with its current implementation, is too inefficient to gain performance improvements over a full-offload deployment (e.g., running `DPFS-Kernel` directly on the DPU). We expect that with the extra cost and energy usage of a gateway deployment with `DPFS-Gateway`, the best option for deploying DPFS++ with Ceph is to run `DPFS-Kernel` and the CephFS client directly on the DPU.



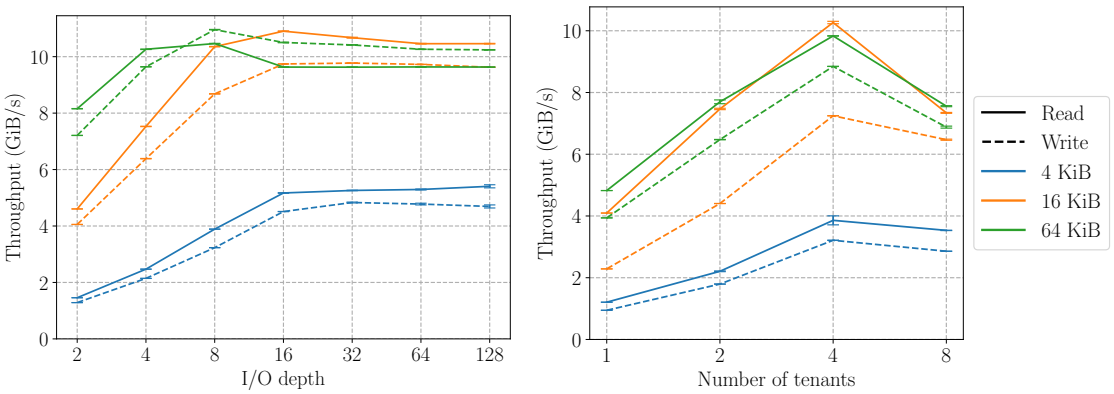
(a) Random 4 KiB I/O throughput

(b) Random 64 KiB I/O throughput

Figure 6.6: Synthetic workload (fio) throughput experiments with Ceph of `DPFS-Gateway` and the `DPFS-Kernel` backend, compared against host-NFS and `DPFS-Kernel` on the DPU.

6.5 Multi-tenancy in DPFS++

The last set of experiments in this chapter is aimed at evaluating the multi-tenancy support of DPFS++. For the purpose of this evaluation, three extra experimental variables are introduced: the number of tenants (i.e., number of fio clients), the number of `virtio-fs` device poller threads (see Section 5.4) and the number of `io_uring` completion polling threads (see Section 5.4.2). It is important to note that in all the multi-tenancy experiments, every `virtio-fs` device (i.e., every tenant) has a queue depth of 256, instead of the 512 queue depth used in Section 4.3. We made this change because, when experimenting with eight tenants, the memory consumption of the NVIDIA SNAP library (see Section 4.2.3) with eight `virtio-fs` devices and 512 descriptors is greater than the 8 GiB memory capacity of the NVIDIA BlueField-2 DPU.



(a) Combined 4 KiB, 16 KiB and 64 KiB random I/O throughput of `DPFS-Null` with 8 tenants and 4 `virtio-fs` device poller threads.

(b) Combined 4 KiB, 16 KiB and 64 KiB random I/O (fixed I/O depth at 128) throughput of `DPFS-Null` with 1,2,4,8 tenants and 2 `virtio-fs` device poller threads.

Figure 6.7: Synthetic workload (fio) throughput experiments of `DPFS-Null` with multi-tenancy.

In Section 6.5.1, the `DPFS-Null` backend is used to measure the best performance the NVIDIA BlueField-2 can provide regarding throughput and latency. To evaluate the performance of multi-tenancy under in vitro cloud data center conditions, we measure the throughput of `DPFS-Kernel` when connected to the Ceph cluster described in Section 6.5.2.

6.5.1 Finding the limit of the NVIDIA BlueField-2's hardware

To test the NVIDIA BlueField-2's limits in a multi-tenancy deployment, we perform several experiments on `DPFS-Null` with differing numbers of `virtio-fs` poller threads and tenants. Figure 6.7 summarizes the results that best showcase the performance characteristics of the NVIDIA BlueField-2 DPU. The standard deviation shown in these two graphs, is the standard deviation of the list of average throughput per tenant. Thus showing the difference in performance between tenants.

We find that with four `virtio-fs` device poller threads and eight tenants (i.e., fio instances), shown in Figure 6.7a, we can achieve the highest random I/O throughput at 1.4 million read and 1.2 million write 4 KiB IOPS at I/O depth 128. Whereas eight `virtio-fs` poller threads, achieve 1.36 million read and 1.1 million write IOPS. Even though with eight poller threads, every `virtio-fs` has its own dedicated poller thread, the 8-core ARM CPU of the NVIDIA BlueField-2 is overwhelmed by the eight busy polling threads. With the 16 KiB and 64 KiB block sizes, the total throughput that the DPU can deliver tops out at 11.1 GiB/s (for reads at I/O depth 16 and writes at I/O depth 8). The theoretical total

throughput that the DPU can deliver to the host CPU is 14.7 GiB/s through its PCIe 3.0 x16 connection. The `virtio-fs` throughput we measure is roughly 3/4 of the DPU's maximum PCIe throughput.

6.5.2 Ceph multi-tenancy storage using `DPFS-Kernel`

The last experiment of this chapter evaluates the behavior of `DPFS-Kernel` when connected to the Ceph cluster (Section 6.3.2) in a multi-tenancy cloud-like deployment of DPFS++. Its results do not warrant figures, as the multi-tenancy performance is virtually the same as the single-tenant performance. Using four `virtio-fs` poller threads, three `io_uring` completion polling threads, and eight `virtio-fs` devices / tenants, 4 KiB random reads give a combined throughput (across all eight tenants) of 147.5 MiB/s at an I/O depth of 128. In our previous single-tenant experiment, we measured 129 MiB/s of read throughput for the same workload. A 14% increase in throughput while the parallelism increases by roughly 400% indicates that there is a large bottleneck somewhere in `DPFS-Kernel` or in the connection to the remote Ceph storage cluster. Investigating the bottleneck, we find the bottleneck to be the single CephFS mount in the Linux kernel that is mirrored to all the tenants. This is indicated by the observation that we see the same performance characteristics when running fio on the DPU's Linux operating system (no DPFS++ or host involved) with the CephFS file system client. The inability of the CephFS file system client to handle parallelism is a known performance issue that can be remedied by creating multiple CephFS mounts (38). For optimal performance, `DPFS-Kernel` implementation should therefore be modified to give every tenant their own file system mount that resides in the Linux kernel.

7

Concluding remarks

The impact of data-driven systems running in cloud data centers reaches into virtually every aspect of our daily lives. The massive amounts of energy required to power these cloud data centers present the research community with two critical objectives: performance and efficiency. Novel hardware architectures with heterogeneous computing and offloading capabilities, like DPUs (programmable offload capable networking cards), have become ubiquitous in our cloud data centers. In this thesis, we expanded the current body of research on distributed file system storage powered by DPUs. We break down the currently explored design space and propose new system designs that can be implemented with current DPU hardware. Using the findings of the previously published research prototype DPFS (51), we design, implement, and evaluate (1) performance improvements to the `virtio-fs` driver, (2) support for kernel-based distributed file systems using `io_uring`, (3) implement the newly proposed gateway design and (4) support for multi-tenancy.

In this chapter, we summarize and conclude our findings by answering the research questions that motivated this research in Section 7.1, summarize the current limitations of DPFS++ in Section 7.2, and establish research directions that future research works can take in Section 7.3.

7.1 Conclusion

In Chapter 1, we described the problem statement and defined research questions to alleviate the issues that DPFS faces. Using the results found through design, implementation, and evaluation methods in Chapter 4, Chapter 5, and Chapter 6, we will draw conclusions and answer the research questions.

With these findings, we conclude that our proposed DPFS++ framework shows that DPU-powered file system offloading through our framework is cloud-capable. We are able to connect multiple tenants on the host to a remote Ceph cluster via the Nvidia BlueField-2, and we report that its performance is on par with the systems currently deployed in the cloud. However, a cloud-native solution with state-of-the-art performance and dynamic multi-tenancy requires significant further efforts in research and engineering.

RQ1: How can the current generation of DPU hardware facilitate file system offloading? In Chapter 3, we categorize the existing DPU-offloaded DFS literature into two designs: partial offload (LineFS (74) and Di Girolamo et al. (44)) and guided passthrough (Fisc (51)). On top of those, we define three new designs: (1) full offload, (2) gateway passthrough, and (3) decoupling, and further identify the possibilities of implementing these designs using the hardware offloading capabilities available in the current generation of DPUs (e.g., FPGA and DSL offloading engines). The full offload and decoupling designs are utilized in the design of DPFS (Chapter 4).

We implement the gateway passthrough design in DPFS++ using eRPC, and our evaluation finds that it can only support 114k 4 KiB IOPS that terminate at the gateway compared to 327k 4 KiB IOPS that end at the DPU. When running `DPFS-Kernel` on the gateway, mirroring a Ceph storage cluster, we report that it performs equal (within 6%) or significantly worse (up to 5.5×) than running the `DPFS-Kernel` mirror on the DPU itself.

RQ2: How can the performance of the Linux kernel be improved for DPU-offloaded file systems? We find that the `virtio-fs` device driver in the Linux kernel poorly handles deep I/O depths (resulting in high tail latencies) and is not well adjusted to multi-core CPUs and DPUs. To remedy these issues, we design and implement two patches for the `virtio-fs` device driver. We evaluate the first patch that alleviates the deep I/O depth latency issue and find that it reduces average latencies by 3-5% and tail latencies by more than two orders of magnitude in Section 6.2.

RQ3: How can DPFS efficiently utilize existing kernel-based distributed file systems? To support existing kernel-based (distributed) file systems that are currently widely deployed in many data centers, in DPFS++. We propose to leverage the `io_uring` Linux API as it is the best candidate for accessing the file systems from userspace. DPFS++ includes the new file system backend `DPFS-Kernel` that mirrors (using `io_uring`) a file system mounted in the DPU's Linux kernel to the host CPU. Our evaluation reports that `DPFS-Kernel` can support up to 154k 4 KiB IOPS to a RAM-backed file system in the kernel. When `DPFS-Kernel` mirrors a Ceph remote storage cluster, and running synthetic

and metadata workloads on the host, we find that overall, it performs on par with the host NFS approach used in current cloud data centers.

RQ4: How can DPFS efficiently support dynamic multi-tenancy? For dynamic data centers like cloud data centers, dynamically scaling a storage service is crucial. We identify that to support dynamic multi-tenancy in DPFS++, support for SR-IOV by the DPU is required. However, while the NVIDIA BlueField-2 technically supports SR-IOV, it does not function correctly with `virtio-fs`. So we implement multi-tenancy support in the DPFS++ framework using physical functions that are defined at hardware boot and only support up to ten `virtio-fs` devices (i.e., tenants). It is resulting in DPFS++ not being able to dynamically scale the number of tenants.

To efficiently support the polling on multiple `virtio-fs` devices, we identify that the scheduler of SPDK can be re-used in DPFS++. However, we only implement a simpler static scheduler in DPFS++ that does not scale the DPUs resources depending on the workload hitting the `virtio-fs` devices. Our evaluation shows that the NVIDIA BlueField-2 can support up to 1.4 million 4 KiB IOPS and more than 10GiB/s of throughput. Using the `DPFS-Kernel` backend in a multi-tenancy deployment, we report that its implementation can only provide 14% more throughput in total for Eight tenants, compared to a single-tenant setup. Thus, we conclude that the implementation poorly suits multi-tenancy workloads and performs significantly worse than the host NFS approach.

7.2 Limitations

As a result of the implementation and evaluation of DPFS++, we identify two major limitations to the work that are described in this section.

`DPFS-Kernel` multi-tenancy optimizations

Our evaluation finds that when utilizing the `DPFS-Kernel` backend in a multi-tenancy deployment, mirroring a single Ceph file system folder to the many tenants on the host results in poor performance. Accessing a single Ceph file system from multiple cores on the DPU creates a bottleneck because of locking tension in the Ceph file system client. We expect this bottleneck can be alleviated by mounting a file system per tenant, giving every tenant their DFS client running on the DPU.

Dynamic multi-tenancy (scheduler and SR-IOV) not implemented and evaluated

The multi-tenancy design in DPFS++ encompasses a dynamic scheduler for polling the `virtio-fs` devices, aiming to save energy by putting the CPU in low-power mode when the system is under low load. It also contains a method for dynamically changing the `virtio-fs` devices the DPU exposes through the SR-IOV PCIe protocol. These two design elements were not implemented nor evaluated to limit the scope of the thesis (dynamic scheduler) and because of hardware/software instability issues (SR-IOV). Because of this, the experiments in this thesis could only evaluate DPFS++ in a static cloud-like environment (in vitro), as opposed to the dynamic environment in real-world cloud data centers.

7.3 Future work

To pave the way for further advancements in this area of research, we propose two research directions that future work can take to build upon the findings of this thesis:

Exploiting the DPU's hardware offloading capabilities

DPUs contain various offloading engines specialized to multiple application domains such as security, networking, and storage. We think these engines could be utilized for implementing a DFS client, for example, employing the encryption and decryption engines commonly found in DPUs to secure the contents of file system reads and writes. This could improve performance over a software-based encryption mechanism and enhance data security by encrypting the data that later gets stored by the remote storage backend.

DPU-native distributed file system - alleviating the memory bottleneck

A recurring theme throughout the conclusion of this thesis is the bottleneck that data copies on the DPU create. In every backend of DPFS++ (e.g., `DPFS-Gateway`, `DPFS-Kernel`, `DPFS-NFS`), we identify that data copies significantly reduce the performance of the backends. We think a promising approach for removing these data copies is direct RDMA from the host memory to the remote storage backend. DPUs like the NVIDIA BlueField-2 are capable of RDMA-ing host memory to a remote server without the data being copied into the DPU's memory, evading the DPU's low memory bandwidth. To implement this, the DFS must be tightly integrated into the complete software stack on the DPU, requiring significant software engineering effort. However, we expect such an implementation to

perform much closer to the maximum bandwidth of a DPU-based `virtio-fs` device than DPFS++ is.

References

[1] **Erasure Code — Ceph Documentation**. Available from: https://docs.ceph.com/en/latest/rados/operations/erasure-code/. 19

[2] **NFS-Ganesha - Userspace NFS Server**. Available from: https://github.com/nfs-ganesha/nfs-ganesha. 19

[3] **Libfuse: The Reference Implementation of the Linux FUSE (Filesystem in Userspace) Interface**, 2023. Available from: https://github.com/libfuse/libfuse. 34, 37, 49

[4] Achronix. **VectorPath S7t-VG6 Accelerator Card**, April 2023. Available from: https://www.achronix.com/sites/default/files/docs/Vectorpath_S7t-VG6_Accelerator_Card_Bittware_PB999_V4.pdf. 23

[5] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. **Firecracker: Lightweight Virtualization for Serverless Applications**. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020. Available from: https://www.usenix.org/conference/nsdi20/presentation/agache. 32

[6] Amazon AWS. **What Is Amazon Elastic File System? - Amazon Elastic File System**, 2023. Available from: https://docs.aws.amazon.com/efs/latest/ug/whatisefs.html. 3, 19, 33

[7] AMD. **AMD EPYC™ 9754 Specifications**. Available from: https://www.amd.com/en/product/13371. 6

[8] AMD. **AMD Expands Data Center Solutions Capabilities with Acquisition of Pensando**, April 2022.

Available from: https://www.amd.com/en/press-releases/2022-04-04-amd-expands-data-center-solutions-capabilities-acquisition-pensando. 17

[9] AMD. **Data Center Networking**, 2023. Available from: https://www.xilinx.com/applications/data-center/network-acceleration.html. 22, 29

[10] Nadav Amit, Michael Wei, and Dan Tsafrir. **Dealing with (Some of) the Fallout from Meltdown**. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR '21, pages 1–6, New York, NY, USA, June 2021. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3456727.3463776. 52

[11] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. **Above the Clouds: A Berkeley View of Cloud Computing**. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009. Available from: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html. 1, 5, 11

[12] Amazon AWS. **The Security Design of the AWS Nitro System - AWS Whitepaper**, November 2022. Available from: https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf. 2, 3, 16, 17, 21, 23, 61

[13] Amazon AWS. **Block vs File vs Object Storage - Difference Between Data Storage Services - AWS**, 2023. Available from: https://aws.amazon.com/compare/the-difference-between-block-file-object-storage/. 3

[14] Jens Axboe. **Flexible I/O Tester**, 2022. Available from: https://github.com/axboe/fio. 66

[15] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish

Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. **Empowering Azure Storage with RDMA**. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association. Available from: `https://www.usenix.org/conference/nsdi23/presentation/bai`. 20, 57

[16] Oana Balmau. **Characterizing I/O in Machine Learning with MLPerf Storage**. *ACM SIGMOD Record*, **51**(3):47–48, November 2022. Available from: `https://dl.acm.org/doi/10.1145/3572751.3572765`. 2, 36

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. **Xen and the Art of Virtualization**. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, October 2003. Association for Computing Machinery. Available from: `http://doi.org/10.1145/945445.945462`. 12

[18] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. **Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems**. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 1–10, New York, NY, USA, June 2013. Association for Computing Machinery. Available from: `https://doi.org/10.1145/2485732.2485740`. 49

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. **P4: Programming Protocol-Independent Packet Processors**. *ACM SIGCOMM Computer Communication*

Review, **44**(3):87–95, July 2014. Available from: https://dl.acm.org/doi/10.1145/2656877.2656890. 24

[20] Andrew Boutros and Vaughn Betz. **FPGA Architecture: Principles and Progression**. *IEEE Circuits and Systems Magazine*, **21**(2):4–29, 2021. Available from: https://ieeexplore.ieee.org/document/9439568. 22

[21] Broadcom. **Product Brief: Stingray PS225**, 2018. Available from: https://web.archive.org/web/20221222194957/https://docs.broadcom.com/doc/PS225-PB. 23

[22] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. **State-of-the-Art in Heterogeneous Computing**. *Scientific Programming*, **18**(1):1–33, January 2010. Available from: https://content.iospress.com/articles/scientific-programming/spr296. 15

[23] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. **Language Models Are Few-Shot Learners**, July 2020. Available from: http://arxiv.org/abs/2005.14165. 2

[24] Emilio J. Bueno, Alvaro Hernandez, Francisco J. Rodriguez, Carlos Giron, RaÚl Mateos, and Santiago Cobreces. **A DSP- and FPGA-Based Industrial Control With High-Speed Communication Interfaces for Grid Converters Applied to Distributed Power Generation Systems**. *IEEE Transactions on Industrial Electronics*, **56**(3):654–669, March 2009. Available from: https://ieeexplore.ieee.org/document/4663812. 22

[25] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. **Bringing Virtualization to the X86 Architecture with the Original VMware Workstation**. *ACM Transactions on Computer Systems*, **30**(4):12:1–12:51, November 2012. Available from: https://dl.acm.org/doi/10.1145/2382553.2382554. 12

[26] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. **Intel's Hyperscale-Ready Infrastructure Processing Unit (IPU)**. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–16, Palo Alto, CA, USA, August 2021. IEEE. Available from: `https://ieeexplore.ieee.org/document/9567455/`. 22, 23, 25, 60, 61

[27] Idan Burstein. **Nvidia Data Center Processing Unit (DPU) Architecture**. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20, Palo Alto, CA, USA, August 2021. IEEE. Available from: `https://ieeexplore.ieee.org/document/9567066/`. 16, 18

[28] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. **Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency**. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, October 2011. Association for Computing Machinery. Available from: `http://doi.org/10.1145/2043556.2043571`. 19

[29] Maria Carla Calzarossa, Luisa Massari, and Daniele Tessera. **Workload Characterization: A Survey Revisited**. *ACM Computing Surveys*, **48**(3):48:1–48:43, February 2016. Available from: `https://dl.acm.org/doi/10.1145/2856127`. 5

[30] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. **Characterizing, Modeling, and Benchmarking {RocksDB} {Key-Value} Workloads at Facebook**. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020. Available from: `https://www.usenix.org/conference/fast20/presentation/cao-zhichao`. 66

[31] Hung-Hsin Chen, Chih-Hao Chang, and Shih-Hao Hung. **hKVS: A Framework for Designing a High Throughput Heterogeneous Key-Value Store with SmartNIC and RDMA**. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, pages 99–106, Virtual Event Japan, October 2022. ACM. Available from: https://dl.acm.org/doi/10.1145/3538641.3561495. 3

[32] Google Cloud. **Object Storage vs Block Storage vs File Storage: Which Should You Choose?**, May 2021. Available from: https://cloud.google.com/blog/topics/developers-practitioners/map-storage-options-google-cloud. 3, 18

[33] Google Cloud. **Introducing C3 Machines with Google's Custom Intel IPU**, October 2023. Available from: https://cloud.google.com/blog/products/compute/introducing-c3-machines-with-googles-custom-intel-ipu. 3, 17

[34] Google Cloud. **Titanium Underpins Google's Workload-Optimized Infrastructure**, August 2023. Available from: https://cloud.google.com/blog/products/compute/titanium-underpins-googles-workload-optimized-infrastructure. 23

[35] IBM Cloud. **Object vs. File vs. Block Storage: What's the Difference?**, October 2021. Available from: https://www.ibm.com/blog/object-vs-file-vs-block-storage/. 3

[36] Jonathan Corbet. **The Rapid Growth of Io_uring**, January 2020. Available from: https://lwn.net/Articles/810414/. 52

[37] Steve Crago, Kyle Dunn, Patrick Eads, Lorin Hochstein, Dong-In Kang, Mikyung Kang, Devendra Modium, Karandeep Singh, Jinwoo Suh, and John Paul Walters. **Heterogeneous Cloud Computing**. In *2011 IEEE International Conference on Cluster Computing*, pages 378–385, September 2011. Available from: https://dl.acm.org/doi/10.1109/CLUSTER.2011.49. 13

[38] croit GmbH. **Achieving maximum performance from a fixed size Ceph object storage cluster**, May 2021. Available from: https://www.redhat.com/en/blog/achieving-maximum-performance-fixed-size-ceph-object-storage-cluster. 77

[39] Jad Darrous and Shadi Ibrahim. **Understanding the Performance of Erasure Codes in Hadoop Distributed File System**. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '22, pages 24–32, New York, NY, USA, April 2022. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3503646.3524296. 19

[40] Jaideep Dastidar, David Riddoch, Jason Moore, Steven Pope, and Jim Wesselkamper. **The AMD 400-G Adaptive SmartNIC System on Chip: A Technology Preview**. *IEEE Micro*, **43**(3):40–49, May 2023. Available from: https://ieeexplore.ieee.org/document/10078398. 22, 23, 25

[41] Benoît Dupont de Dinechin. **Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore Processor**. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 1–4, New York, NY, USA, June 2019. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3316781.3323473. 23, 24

[42] Christina Delimitrou and Christos Kozyrakis. **Bolt: I Know What You Did Last Summer... In The Cloud**. *ACM SIGARCH Computer Architecture News*, **45**(1):599–613, April 2017. Available from: https://dl.acm.org/doi/10.1145/3093337.3037703. 32, 33

[43] L. Peter Deutsch and Allan M. Schiffman. **Efficient Implementation of the Smalltalk-80 System**. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, January 1984. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/800017.800542. 12

[44] Salvatore Di Girolamo, Daniele De Sensi, Konstantin Taranov, Milos Malesevic, Maciej Besta, Timo Schneider, Severin Kistler, and Torsten Hoefler. **Building Blocks for Network-Accelerated Distributed File Systems**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, pages 1–14, Dallas, Texas, November 2022. IEEE Press. Available from: https://ieeexplore.ieee.org/abstract/document/10046100. 25, 26, 80

[45] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. **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*, SYSTOR '22, pages 120–127, New York, NY, USA, June 2022. Association for Computing Machinery. Available from: https://doi.org/10.1145/3534056.3534945. 39, 52

[46] Nick Feamster, Jennifer Rexford, and Ellen Zegura. **The Road to SDN: An Intellectual History of Programmable Networks**. *ACM SIGCOMM Computer Communication Review*, **44**(2):87–98, April 2014. Available from: http://doi.org/10.1145/2602204.2602219. 13

[47] 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. **Azure Accelerated Networking: SmartNICs in the Public Cloud**. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 51–64, USA, April 2018. USENIX Association. Available from: https://www.usenix.org/conference/nsdi18/presentation/firestone. 2, 13, 16, 17, 23, 34

[48] IO500 Foundation. **IO500 - ISC23 - Full List**, 2023. Available from: https://io500.org/list/isc23/full. 45, 52, 69

[49] 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. **When Cloud Storage Meets {RDMA}**. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021. Available from: https://www.usenix.org/conference/nsdi21/presentation/gao. 20, 57

[50] Google Cloud. **Technical Overview - Google Cloud Filestore**, 2023. Available from: https://cloud.google.com/filestore/docs/overview. 3, 19, 33

[51] Peter-Jan Gootzen, Jonas Pfefferle, Radu Stoica, and Animesh Trivedi. **DPFS: DPU-Powered File System Virtualization**. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, SYSTOR '23, pages 1–7, New York, NY, USA, June 2023. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3579370.3594769. iii, 4, 8, 9, 19, 39, 79, 80

[52] Karan Gupta. **From Hyper Converged Infrastructure to Hybrid Cloud Infrastructure**, 2020-Jul. Available from: https://www.usenix.org/conference/hotstorage20/presentation/keynote-gupta. 12

[53] Richard R Hamming. *Art of Doing Science and Engineering: Learning to Learn*. CRC Press, 1997. 7

[54] Thomas Haynes. **Network File System (NFS) Version 4 Minor Version 2 Protocol**. Request for Comments RFC 7862, Internet Engineering Task Force, November 2016. Available from: https://datatracker.ietf.org/doc/rfc7862. 19

[55] Gernot Heiser. **Gernot's List of Systems Benchmarking Crimes**, 2023. Available from: https://gernot-heiser.org/benchmarking-crimes.html. 7

[56] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, October 2011. 15

[57] John L. Hennessy and David A. Patterson. **A New Golden Age for Computer Architecture**. *Communications of the ACM*, **62**(2):48–60, January 2019. Available from: https://dl.acm.org/doi/10.1145/3282307. 2, 15, 23

[58] Dean Hildebrand, Arifa Nisar, and Roger Haskin. **pNFS, POSIX, and MPI-IO: A Tale of Three Semantics**. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 32–36, New York, NY, USA, November 2009. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/1713072.1713082. 20

[59] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. **XFUSE: An Infrastructure for Running Filesystem Services in**

User Space. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 863–875, 2021. Available from: https://www.usenix.org/conference/atc21/presentation/hsu. 49

[60] Huawei. **Computing 2030**, 2021. Available from: https://www-file.huawei.com/-/media/corp2020/pdf/giv/industry-reports/computing_2030_en.pdf. 1

[61] IBM Cloud. **IBM Cloud File Storage - Overview**, July 2022. Available from: https://www.ibm.com/cloud/file-storage. 3, 19

[62] Primate Labs Inc. **Geekbench 6 - Cross-Platform Benchmark**, 2023. Available from: https://www.geekbench.com/. 55

[63] Intel. **SPDK: Scheduler**, October 2022. Available from: https://spdk.io/doc/scheduler.html. 61

[64] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, May 2022. Available from: http://arxiv.org/abs/2206.03259. 1, 9

[65] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776, July 2019. 7

[66] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Measurement, Simulation, and Modeling*. john wiley & sons, 2008. 7

[67] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. **TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings**, April 2023. Available from: http://arxiv.org/abs/2304.01433. 15

[68] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. **In-Datacenter Performance Analysis of a Tensor Processing Unit**. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, June 2017. Association for Computing Machinery. Available from: `https://dl.acm.org/doi/10.1145/3079856.3080246`. 15

[69] Anuj Kalia, Michael Kaminsky, and David Andersen. **Datacenter {RPCs} Can Be General and Fast**. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019. Available from: `https://www.usenix.org/conference/nsdi19/presentation/kalia`. 57

[70] Anuj Kalia, Michael Kaminsky, and David G. Andersen. **Design Guidelines for High Performance {RDMA} Systems**. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, 2016. Available from: `https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia`. 49

[71] Kalray. **Kalray's MPPA® DPU Processor (Data Processing Unit)**, August 2023. Available from: `https://www.kalrayinc.com/products/processors-many-core/`. 24, 25

[72] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. **Profiling a Warehouse-Scale Computer**. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, June 2015. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/2749469.2750392. 2

[73] Patrick Kennedy. **DPU vs SmartNIC and the STH NIC Continuum Framework**, May 2021. Available from: https://www.servethehome.com/dpu-vs-smartnic-sth-nic-continuum-framework-for-discussing-nic-types/. 16

[74] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. **LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism**. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 756–771, New York, NY, USA, October 2021. Association for Computing Machinery. Available from: https://doi.org/10.1145/3477132.3483565. 3, 19, 25, 28, 32, 80

[75] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. **Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors**. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014. Available from: https://ieeexplore.ieee.org/document/6853210. 14

[76] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. **ReFlex: Remote Flash ≈ Local Flash**. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, New York, NY, USA, April 2017. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3037697.3037732. 32

[77] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas

Prescher, Michael Schwarz, and Yuval Yarom. **Spectre Attacks: Exploiting Speculative Execution**. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019. Available from: https://ieeexplore.ieee.org/document/8835233. 14

[78] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. **Understanding RDMA Microarchitecture Resources for Performance Isolation**. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 31–48, Boston, MA, April 2023. USENIX Association. Available from: https://www.usenix.org/conference/nsdi23/presentation/kong. 20, 57

[79] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. **Modernizing File System through In-Storage Indexing**. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 75–92, 2021. Available from: https://www.usenix.org/conference/osdi21/presentation/koo. 28, 35

[80] Nectarios Koziris. **Fifty Years of Evolution in Virtualization Technologies: From the First IBM Machines to Modern Hyperconverged Infrastructures**. In *Proceedings of the 19th Panhellenic Conference on Informatics*, PCI '15, pages 3–4, New York, NY, USA, October 2015. Association for Computing Machinery. Available from: http://doi.org/10.1145/2801948.2802039. 12, 13

[81] Ian Kuon and Jonathan Rose. **Measuring the Gap between FPGAs and ASICs**. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, February 2006. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/1117201.1117205. 22

[82] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. **The Open Images Dataset V4: Unified Image Classification, Object Detection, and Visual Relationship Detection at Scale**. *IJCV*, 2020. 2

[83] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. **Atlas: Baidu's Key-Value Storage System for Cloud Data**. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, May 2015. 33

[84] Christian Leber, Benjamin Geib, and Heiner Litz. **High Frequency Trading Acceleration Using FPGAs**. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 317–322, September 2011. 22

[85] Yair Levy and Timothy J Ellis. **A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research.** *Informing Science*, **9**, 2006. 7

[86] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. **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*, ASPLOS '20, pages 591–605, New York, NY, USA, March 2020. Association for Computing Machinery. Available from: `https://doi.org/10.1145/3373376.3378531`. 18, 33

[87] 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. **Fisc: A Large-scale Cloud-native-oriented File System**. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 231–246, 2023. Available from: `https://www.usenix.org/conference/fast23/presentation/li-qiang-fisc`. 16, 17, 19, 25, 26, 32

[88] Kunal Lillaney, Vasily Tarasov, David Pease, and Randal Burns. **Agni: An Efficient Dual-access File System over Object Storage**. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 390–402, New York, NY, USA, November 2019. Association for Computing Machinery. Available from: `https://dl.acm.org/doi/10.1145/3357223.3362703`. 33

[89] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. **Meltdown: Reading Kernel Memory from User Space**. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018. Available from: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp. 14

[90] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. **Offloading Distributed Applications onto smartNICs Using iPipe**. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 318–333, New York, NY, USA, August 2019. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3341302.3342079. 3

[91] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. **E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers**. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019. Available from: https://www.usenix.org/conference/atc19/presentation/liu-ming. 3, 61

[92] Redis Ltd. **Redis**, 2023. Available from: https://github.com/redis/redis. 66

[93] Youyou Lu, Jiwu Shu, and Weimin Zheng. **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*, FAST'13, pages 257–270, USA, February 2013. USENIX Association. Available from: https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou. 33

[94] Layong (Larry) Luo. **Towards Converged SmartNIC Architecture for Bare Metal & Public Clouds**, August 2018. Available from: https://conferences.sigcomm.org/events/apnet2018/slides/larry.pdf. 16, 23

[95] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. **InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems**. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 313–328, 2022. Available from: https://www.usenix.org/conference/fast22/presentation/lv. 32, 52

[96] Marvell. **Marvell® OCTEON 10 DPU Platform**, 2021. Available from: https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-octeon-10-dpu-platform-product-brief.pdf. 23

[97] Francis Matus. **Distributed Services Architecture**. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–17, Palo Alto, CA, USA, August 2020. IEEE. Available from: https://ieeexplore.ieee.org/document/9220629/. 2, 18, 24, 29, 60, 61

[98] Max Maxfield. **ASIC vs. ASSP vs. SoC vs. FPGA – What's the Difference?**, June 2014. Available from: https://www.eetimes.com/asic-assp-soc-fpga-whats-the-difference/. 22

[99] 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. **From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud**. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 753–766, New York, NY, USA, August 2022. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3544216.3544238. 18, 32

[100] Microsoft. **Windows Server 2022 Licensing & Pricing | Microsoft**. Available from: https://www.microsoft.com/en-us/windows-server/pricing. 13, 18

[101] Microsoft Azure. **Azure Files - Managed File Shares and Storage | Microsoft Azure**, 2022. Available from: https://azure.microsoft.com/en-us/services/storage/files/. 3, 19, 33

[102] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. **Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs**. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 106–122, New York, NY, USA, August 2021. Association for Computing Machinery. Available from: http://doi.org/10.1145/3452296.3472940. 33

[103] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. **Analyzing and Mitigating Data Stalls in DNN Training**. *Proceedings of the VLDB Endowment*, **14**(5):771–784, January 2021. Available from: https://dl.acm.org/doi/10.14778/3446095.3446100. 2, 36

[104] Andrew Moore and Jim Henrys. **IPU Based Cloud Infrastructure: The Fulcrum for Digital Business**. Technical report, Intel, 2021. Available from: https://www.intel.com/content/www/us/en/products/docs/programmable/ipu-based-cloud-infrastructure-white-paper.html. 14, 18, 22

[105] Steve Morgan. **The 2020 Data Attack Surface Report**, 2020. Available from: https://goto.arcserve.com/Global-Ongoing-eBookdataattacksurfacereport_eBookLP.html. 9

[106] Beck Motti. **Doubling Network File System Performance with RDMA-Enabled Networking**, March 2021. Available from: https://developer.nvidia.com/blog/doubling-network-file-system-performance-with-rdma-enabled-networking/. 20

[107] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. **Non-Volatile Storage: Implications of the Datacenter's Shifting Center**. *Queue*, **13**(9):33–56, November 2015. Available from: https://dl.acm.org/doi/10.1145/2857274.2874238. 2, 32

[108] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. **HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases**. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, 2017. Available from: https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi. 32, 52

[109] David Noveck and Chuck Lever. **Network File System (NFS) Version 4 Minor Version 1 Protocol**. Request for Comments RFC 8881, Internet Engineering Task Force, August 2020. Available from: https://datatracker.ietf.org/doc/rfc8881. 20, 38, 58

[110] NVIDIA. **VirtIO-net Emulated Devices - BlueField DPU BSP 4.0.3 - NVIDIA Networking Docs**. Available from: https://docs.nvidia.com/networking/display/BlueFieldDPUOSLatest/VirtIO-net+Emulated+Devices#heading-Virtio-netSR-IOVVFDevices. 62

[111] NVIDIA. **NVIDIA BlueField-2 DPU Datasheet**, 2021. Available from: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. 23, 25, 31, 34, 44, 49, 60, 61, 62

[112] NVIDIA. **NVIDIA BlueField-3 DPU Datasheet**, 2021. Available from: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. 23, 24, 25, 49

[113] NVIDIA. **Hands-on Access to VMware vSphere on NVIDIA BlueField DPUs with NVIDIA LaunchPad**, December 2022. Available from: https://developer.nvidia.com/blog/hands-on-access-to-vmwares-vsphere-on-nvidia-bluefield-dpus-with-nvidia-launchpad/. 18, 61

[114] NVIDIA. **NVIDIA CONNECTX-7 | Datasheet**, 2023. Available from: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf. 16

[115] NVIDIA. **NVIDIA Converged Accelerators | Datasheet**, 2023. Available from: https://resources.nvidia.com/en-us-accelerated-networking-resource-library/converged-accelerators. 24

[116] NVIDIA. **Oracle Cloud Infrastructure Chooses NVIDIA BlueField Data Center Acceleration Platform**, March 2023. Available from: http://nvidianews.nvidia.com/news/oracle-cloud-infrastructure-chooses-nvidia-bluefield-data-center-acceleration-platform 3

[117] NVIDIA. **XLIO Library Architecture - NVIDIA Accelerated IO (XLIO) Documentation v1.1.8 - NVIDIA Networking Docs**, 2023. Available from: https://docs.nvidia.com/networking/display/XLIOv214. 34, 38

[118] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. **Improving Agility and Elasticity in Bare-metal Clouds**. *ACM SIGPLAN Notices*, **50**(4):145–159, March 2015. Available from: https://dl.acm.org/doi/10.1145/2775054.2694349. 32

[119] John Ousterhout. **Always Measure One Level Deeper**. *Communications of the ACM*, **61**(7):74–83, June 2018. Available from: https://dl.acm.org/doi/10.1145/3213770. 7

[120] 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. **The RAMCloud Storage System**. *ACM Transactions on Computer Systems*, **33**(3):7:1–7:55, August 2015. Available from: http://doi.org/10.1145/2806887. 6, 34, 35, 38

[121] A. Padegs. **System/360 and Beyond**. *IBM Journal of Research and Development*, **25**(5):377–390, September 1981. Available from: https://doi.org/10.1147/rd.255.0377. 12

[122] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. **Cloud Container Technologies: A State-of-the-Art Review**. *IEEE Transactions on Cloud Computing*, **7**(3):677–692, July 2019. Available from: https://ieeexplore.ieee.org/document/7922500. 12

[123] PCI-SIG. **PCI Express Base Specification Rev. 4.0 Version 1.0**, September 2017. Available from: https://members.pcisig.com/wg/PCI-SIG/document/10912?downloadRevision=active. 59

[124] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *Journal of management information systems*, **24**(3):45–77, 2007. 7

[125] AMD Pensando. **Project Monterey and the Future of Data Center Modernization**, August 2022. Available from: https://www.amd.com/en/data-center-blogs/Project-Monterey. 3, 18, 61

[126] Daniel R. Perkins and Richard L. Sites. **Machine-Independent PASCAL Code Optimization**. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '79, pages 201–207, New York, NY, USA, August 1979. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/800229.806970. 12

[127] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. **Going Deeper with Embedded FPGA Platform for Convolutional Neural Network**. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, February 2016. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/2847263.2847265. 22

[128] Inc. Red Hat. **Virtio-Fs Linux Kernel Implementation - Linux Kernel Source Tree v6.4**, 2022. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/fuse/virtio_fs.c?h=v6.2. 33, 43, 44, 50

[129] Felix Richter. **Infographic: Amazon Maintains Lead in the Cloud Market**, August 2023. Available from: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers. 17

[130] Dennis M. Ritchie and Ken Thompson. **The UNIX Time-Sharing System**. *Communications of the ACM*, **17**(7):365–375, July 1974. Available from: http://doi.org/10.1145/361011.361061. 12

[131] Zhenyuan Ruan, Tong He, and Jason Cong. **INSIDER: Designing in-Storage Computing System for Emerging High-Performance Drive**. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 379–394, USA, July 2019. USENIX Association. Available from: https://www.usenix.org/conference/atc19/presentation/ruan. 28

[132] Karl Rupp. **Microprocessor Trend Data**, February 2022. Available from: https://github.com/karlrupp/microprocessor-trend-data. v, 14

[133] Rusty Russell. **Virtio: Towards a de-Facto Standard for Virtual I/O Devices**. *ACM SIGOPS Operating Systems Review*, **42**(5):95–103, July 2008. Available from: https://doi.org/10.1145/1400097.1400108. 33

[134] Ronnie Sahlberg. **Libnfs: NFS Client Library**, June 2023. Available from: https://github.com/sahlberg/libnfs. 38

[135] Scott Schweitzer. **Panel: SmartNIC or DPU, Who Wins?** In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, August 2020. Available from: https://technologyevangelist.co/2020/08/25/smartnics-vs-dpus/. 16

[136] L. H. Seawright and R. A. MacKinnon. **VM/370: A Study of Multiplicity and Usefulness**. *IBM Systems Journal*, **18**(1):4–17, March 1979. Available from: https://doi.org/10.1147/sj.181.0004. 12

[137] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. **The Hadoop Distributed File System**. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010. Available from: https://ieeexplore.ieee.org/document/5496972. 19

[138] Pradeep Sindhu. **The Fungible DPU™: A New Category of Microprocessor for the Data-Centric Era : Hot Chips 2020**. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–25, Los Alamitos, CA, USA, August 2020. IEEE Computer Society. Available from: https://doi.ieeecomputersociety.org/10.1109/HCS49909.2020.9220423. 23, 24, 25

[139] J.E. Smith and Ravi Nair. **The Architecture of Virtual Machines**. *Computer*, **38**(5):32–38, May 2005. Available from: https://ieeexplore.ieee.org/abstract/document/1430629. 12

[140] Netronome Systems. **Netronome NFP-4000 Flow Processor - Product Brief**, 2020. Available from: https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf. 16, 23, 24, 25

[141] Vasily Tarasov, Erez Zadok, and Spencer Shepler. **Filebench: A Flexible Framework for File System Benchmarking**. *USENIX; login*, **41**(1):6–12, 2016. Available from: https://www.usenix.org/publications/login/spring2016/tarasov. 66

[142] Maroun Tork, Lina Maudlej, and Mark Silberstein. **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*, ASPLOS '20, pages 117–131, New York, NY, USA, March 2020. Association for Computing Machinery. Available from: `http://doi.org/10.1145/3373376.3378528`. 3

[143] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. **Llama 2: Open Foundation and Fine-Tuned Chat Models**, July 2023. Available from: `http://arxiv.org/abs/2307.09288`. 2

[144] Animesh Trivedi and Marco Spaziani Brunella. **CPU-free Computing: A Vision with a Blueprint**. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, pages 1–14, New York, NY, USA, June 2023. Association for Computing Machinery. Available from: `https://dl.acm.org/doi/10.1145/3593856.3595906`. 2

[145] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Kornilios Kourtis, Ioannis Koltsidas, and Thomas R. Gross. **FlashNet: Flash/Network Stack Co-Design**. *ACM Transactions on*

Storage, **14**(4):30:1–30:29, December 2018. Available from: https://dl.acm.org/doi/10.1145/3239562. 28, 32

[146] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. **Albis: {High-Performance} File Format for Big Data Systems**. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 615–630, 2018. Available from: https://www.usenix.org/conference/atc18/presentation/trivedi. 32

[147] Michael S. Tsirkin and Cornelia Huck. **Virtual I/O Device (VIRTIO) Version 1.2**, September 2022. Available from: https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html. 29, 33, 49, 60

[148] Max Uritsky, Perry Leong, and Andy Reichard. **Introducing Microsoft Azure Boost Preview**, July 2023. Available from: https://aka.ms/azureboost. 3

[149] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is Big Data Performance Reproducible in Modern Cloud Networks?** In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 513–527, 2020. Available from: https://www.usenix.org/conference/nsdi20/presentation/uta. 7

[150] Jeff Vander Stoep. **Android: Protecting the Kernel**, August 2016. Available from: https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf. 19

[151] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. **To FUSE or Not to FUSE: Performance of User-Space File Systems**. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017. Available from: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor. 37, 49

[152] Bill Venners. **The Java Virtual Machine**. *Java and the Java virtual machine: definition, verification, validation*, 1998. 12

[153] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng,

Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. **SRNIC: A Scalable Architecture for RDMA NICs**. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, Boston, MA, April 2023. USENIX Association. Available from: https://www.usenix.org/conference/nsdi23/presentation/wang-zilong. 20

[154] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. **Ceph: A Scalable, High-Performance Distributed File System**. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, USA, November 2006. USENIX Association. Available from: https://dl.acm.org/doi/10.5555/1298455.1298485. 19, 52, 65, 69

[155] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J. G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A. C. 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. **The FAIR Guiding Principles for Scientific Data Management and Stewardship**. *Scientific Data*, **3**(1):160018, March 2016. Available from: https://www.nature.com/articles/sdata201618. 7

[156] NVM Express Workgroup. **NVM Express® Base Specification, Revision 2.0c**, October 2022. Available from: https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf. 58

[157] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. **Unleashing SmartNIC Packet Processing Performance in P4**. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, pages 1028–1042, New York, NY, USA, September 2023. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3603269.3604882. 3

[158] Lily Yang, Todd A. Anderson, Ram Gopal, and Ram Dantu. **Forwarding and Control Element Separation (ForCES) Framework**. Request for Comments RFC 3746, Internet Engineering Task Force, April 2004. Available from: https://datatracker.ietf.org/doc/rfc3746. 23

[159] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. **SPDK: A Development Kit to Build High Performance Storage Applications**. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, December 2017. Available from: https://ieeexplore.ieee.org/document/8241103. 52, 61

[160] Mohamed Zahran. **Heterogeneous Computing: Here to Stay**. *Communications of the ACM*, **60**(3):42–45, February 2017. Available from: https://dl.acm.org/doi/10.1145/3024918. 2, 15

[161] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. **High-Density Multi-tenant Bare-metal Cloud**. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 483–495, New York, NY, USA, March 2020. Association for Computing Machinery. Available from: https://doi.org/10.1145/3373376.3378507. 17, 18, 32

[162] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. **Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation**

Model Training: Industrial Product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pages 1042–1057, New York, NY, USA, June 2022. Association for Computing Machinery. Available from: https://dl.acm.org/doi/10.1145/3470496.3533044. 2

[163] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. **Octopus+: An RDMA-Enabled Distributed Persistent Memory File System**. *ACM Transactions on Storage*, **17**(3):19:1–19:25, August 2021. Available from: https://dl.acm.org/doi/10.1145/3448418. 19, 52

[164] Xiantao Zxt, Zhengxiao Zx, and Justin Song. **High-Density Multi-tenant Bare-metal Cloud with Memory Expansion SoC and Power Management**. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–18, August 2020. Available from: https://ieeexplore.ieee.org/document/9220447. 14, 16, 17, 18, 23, 26, 61

Appendices

A Support for file system operations in `io_uring`

DPFS++ operation	`io_uring` operation	Kernel version
create	*openat*	5.6
fallocate	*fallocate*	5.6
flush	*fsync*	5.1
fsyncdir	*fsync*	5.1
fsync	*fsync*	5.1
getattr	*statx*	5.6
mkdir	*mkdirat*	5.15
open	*openat*	5.6
read	*read*	5.1
release	*closeat*	5.6
rename	*renameat*	5.11
rmdir	*unlinkat*	5.11
symlink	*symlinkat*	5.15
unlink	*unlinkat*	5.11
write	*write*	5.1
lookup	*n/a*	n/a
closedir	X	n/a
flock	X	n/a
mknod	X	n/a
opendir	X	n/a
readdir	X	n/a
releasedir	X	n/a
setattr	X	n/a
statfs	X	n/a

Table 1: Translation of DPFS++ operations to `io_uring` operations and which Linux kernel version is required to use this `io_uring` operation. `io_uring` operations that are marked with *X* are not supported in the Linux kernel as of September 2023 and Linux v6.6.

The DPFS++ operation *lookup* is not applicable to a direct translation to `io_uring` because it is a higher level metadata operation that consists of multiple POSIX file operations (namely *open*, *stat*, and *close*). Because of this complexity, it is implemented using the synchronous POSIX API in `DPFS-Kernel`.

B Artifact Appendix

B.1 Abstract

This artifact description describes how to set up DPFS++ and reproduce the results as seen in the thesis. We explain how to obtain the software, set up the same benchmarking environment, and do the benchmarking. The software consists of multiple parts: the DPFS++ framework (including its external libraries), the NVIDIA SNAP library (for NVIDIA BlueField-2 support), a patched Linux kernel, Ceph, a set of benchmarking tools, benchmarking automation scripts, and scripts to generate the plots used in this thesis.

The name DPFS++ is used as a milestone name for the current iteration of DPFS, which includes additional features in the framework. These names can, therefore, be used interchangeably in the context of this artifact appendix.

B.2 Artifact check-list (meta-information)

- Program: DPFS++ (`https://github.com/IBM/DPFS/`)
- Compilation: GCC version 9.4.0, CMake version 3.16.3, GNU automake 1.16.1 and Python3 (Python is only used for generating the plots)
- Experiments: Experiments for DPFS++ are part of the main DPFS code repository `https://github.com/IBM/DPFS/tree/master/experiments`.
- Publicly available?: The source code of DPFS++ is publicly available at `https://github.com/IBM/DPFS`. However, the NVIDIA SNAP library and prototype firmware required to use `virtio-fs` on the NVIDIA BlueField-2 DPU are not publicly available as they are under NDA.
- Code licenses: DPFS++ and its experimentation suite (excluding external dependencies) are currently fully licensed under the GNU Lesser General Public License v2.1. There are plans to change this licensing structure of the hardware abstraction layer (`DPFS-HAL`) and the file systems backends (e.g., `DPFS-Kernel`) to a more permissive license like MIT to aid cloud data center adoption.

B.3 Description

How to access

The Github DPFS repository contains the code and experimental suite of DPFS++. The repository is configured to include all the necessary software dependencies except for the

NVIDIA SNAP library (as this library is proprietary and under NDA).

```
$ git clone https://github.com/IBM/DPFS
$ git submodule update --init --recursive
```

B.4 Installation

The DPFS project does currently not contain its own self-contained build system, as it is meant to be integrated into the build system of the NVIDIA SNAP library. The repository contains several Automake files (in each of the `dpfs_*` directories) that each compile a part of the DPFS software stack. These Automake files need to be integrated into the Automake build system of NVIDIA SNAP (`mlnx_snap` to be precise).

The NVIDIA SNAP library consists of two layers, `mlnx-libsnap` and `mlnx-snap`, that need to be built in that order. The library contains build instructions.

To build the external eRPC dependency:

```
$ cmake . -DPERF=on -DTRANSPORT=infiniband -DROCE=on
$ make -j
```

To build the external libnfs dependency:

```
$ ./bootstrap
$ CFLAGS=-O3 ./configure --enable-pthread
$ make -j
$ sudo make install
$ sudo rm /etc/ld.so.cache
$ sudo ldconfig
```

The DPFS code contains the following four conditions that can be configured using the Automake build system:

1. `DPFS_RVFS` - Enables the eRPC implementation of the `DPFS-HAL` interface, resulting in all compiled backends not to use the NVIDIA SNAP `virtio-fs` layer, but the eRPC library to function as a gateway (see Section 5.3.

2. `VNFS_NULLDEV` - Changes the `DPFS-NFS` backend to function as `DPFS-Null` (see Section 4.2).

3. `LATENCY_MEASURING_ENABLED` - Enables the measuring of NFS operation latencies (round-trip) to the NFS server. This only functions when running a single `virtio-fs` device and with a single `virtio` queue (i.e., single-threaded mode).

4. `IORING_DISABLE_METADATA` - Disables the `io_uring` metadata operations in `DPFS-Kernel`, thus forcing all metadata operations to use the synchronous POSIX API. This is useful when running an older Linux kernel (see Appendix A) on the system that runs `DPFS-Kernel` (i.e., a DPU or a gateway).

Hardware and software dependencies

The DPFS++ version of the DPFS framework only supports the NVIDIA BlueField-2 DPU and a prototype firmware is required that enables `virtio-fs` functionality in the firmware of the DPU. Besides the DPU requirement, the other parts of the DPFS++ stack (i.e., the host consuming the `virtio-fs` device, and remote servers running `DPFS-Gateway`, Ceph or an NFS server) do not require "special access" hardware nor firmware.

The DPFS framework, as of DPFS++ depends upon NVIDIA SNAP, specifically `mlnx-libsnap` version 1.3.2 and `mlnx-snap` version 3.5.0. DPFS and NVIDIA SNAP are not plug-and-play; some changes need to be made to the source code of the SNAP library to support `virtio-fs` device emulation via DPFS and accommodate asynchronous functionality in DPFS. We cannot publish these changes or instructions. If one were to get access to the NVIDIA SNAP library and the prototype firmware, the author can be contacted for further details.

The other hardware and software dependencies are specified in Table 6.1 found in the Chapter 6 of the Evaluation.

B.5 Experiment Workflow

Before starting the experiments, the machines used in the experiment must be cleaned of performance anomalies introducing programs running in the background. The `experiments` folder of the DPFS repository contains the `dpu_clean.sh` and `server_clean.sh` scripts that rid the operating system of such processes (of which we are aware, other deployments may have additional processes running that can introduce anomalies). All servers (gateway and Ceph nodes) have their c-states disabled using `setcpulatency.c` program found in `experiments`.

To perform the single tenant host NFS experiments, the `dpu_networking.sh` script must be executed on the DPU, and `host_dpu_networking.sh` on the host. Doing this allows the host to connect the remote NFS gateway server using the following mount command:

```
$ sudo mount -t nfs -o wsize=1048576,rsize=1048576,async $NFS_URI $MNT
```

To perform the single tenant DPFS and DPFS++ experiments, the DPFS framework must be configured using the `conf_example.toml` found in the root of the repository and then started on the DPU with the wanted backend (e.g., `DPFS-Kernel`) as follows:

```
$ sudo ./dpfs_kernel/dpfs_kernel -c ./conf_example.toml
```

Then the host must mount the `virtio-fs` device as follows:

```
$ sudo mount -t virtiofs dpfs-0 $MNT
```

The `experiments/workloads` folder in the DPFS repository contains scripts that run the workloads used in this thesis, namely, fio, rocksdb, redis and filebench. These can be manually run from the host operating system on the `virtio-fs` file system. The `experiments/runners` contains automation scripts that run the workloads for multi-tenancy (that do automatic mounting and unmounting of the NFS and `virtio-fs` file systems), synthetic, metadata (i.e., real-world), and micro-architectural analysis.