D6.1 REPORT ON COMPUTATIONAL STORAGE CAPABILITIES

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.6

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

Report on state-of-the-art techniques for computational storage, near-data processing, and potential side effects in the context of the I/O hierarchy, as well as an overview of automatically determining the capabilities of a storage configuration.

D6.1 REPORT ON COMPUTATIONAL STORAGE CAPABILITIES			
WP6 – Computational Storage			
Type of document	R	Version	1.6
Dissemination level	PU		
Lead partner	ITU		
Author(s)	Philippe Bonnet		
Contributors	all		

Revision History

Version	Item	Comment	Author / Reviewer
V0.1	Structure of the document		Philippe Bonnet
V1.0	First draft	Oct 14th	Philippe Bonnet, Pinar Tözün
V1.1	Second draft	Oct 21st	Philippe Bonnet
V1.3	Third draft	Nov 1st	Marcus Paradies, Philippe Bonnet, Piotr Ratuszniak
V1.4	Almost Complete Draft (missing references, related work and conclusion)	Nov 4th	Philippe Bonnet, Piotr Ratuszniak, Pinar Tözün
V1.5	Draft sent to reviewers	Nov 9th	all
V1.6	Addressed comments from reviewers	Nov 30th	Philippe Bonnet

Executive Summary

This report describes how compute and storage interact within storage devices and across a cluster. The report focuses on the recent evolutions in storage architectures and programming abstractions. In particular, the report defines computational storage and focuses on the motivation for its introduction, the relevant measurables, the architecture of existing devices and the associated programming techniques.

Table of Contents

Table of Figures

List of Abbreviations

Abbreviation	Meaning
SSD	Solid-State Drive
HDD	Hard-Disk Drive
CSP	Computational Storage Processor
HPC	High-Performance Computing
AI	Artificial Intelligence
HPDA	High-Performance Data Analytics
FPGA	Field-Programmable Gate Array
MPSoC	Multi-Processor System on a Chip
DPU	Data Processing Unit

1 Introduction

Integrated data pipelines rely on stored data: shared data sets archived on cold storage, data sets larger than memory during processing, or data stored on high-performance Solid-State Drives (SSDs) rather than memory for better cost/performance.

The DAPHNE system must make decisions about (i) when and where to store data and (ii) how to access and process stored data (assuming appropriate storage resources are provisioned). These decisions depend on the capabilities of the available storage architecture and storage systems. They are not trivial due to the complex and rapidly changing nature of the storage landscape.

First, storage devices have undergone a radical evolution with the advent of NVMe Solid-State Drives. Such SSDs are orders of magnitude faster than the magnetic hard disk drives that were the secondary storage of choice for decades.

Second, storage architectures are evolving to keep up with the demands of the varied workloads that characterize integrated data pipelines, including modeling and simulation (HPC), machine learning and deep learning (AI) and high-performance data analytics (HPDA).

Third, fundamental trends are challenging the traditional separation between compute and storage, based on data movement. Advances in silicon technologies no longer lead to exponential improvements in processor performance. Also, the increase in dataset size leads to increased data movement. As a result, processors and networks are becoming bottlenecks. To address these bottlenecks, public clouds and HPC clusters are deploying computational storage. Computational storage denotes programmable disks, as opposed to traditional disks that are pre-programmed by vendors to read and write data. Computational storage is a solution to the processor and network bottlenecks as data can be processed where it is stored, with minimal data movement. Computational storage can be programmed statically, or run code offloaded dynamically, to meet the needs of a particular data pipeline.

I/Os[1] are issued by a computer to access and store data[2]. With the advent of computational storage, I/Os are also issued to process stored data. In this report, we first survey the I/O workloads associated to the HPC, AI and HPDA components of integrated data pipelines.

We then survey the storage landscape from two viewpoints, relevant for the design, implementation, evaluation, and operation of the DAPHNE run-time:

- The architecture viewpoint: we survey architectures combining storage and compute components including storage devices and distributed storage systems, with a special focus on computational storage.

1 Throughout the document, we denote a transfer between computer and storage device as an I/O. Traditionally, an I/O is a read or a write operation resulting in the transfer of data to or from the host.
2 Our focus on I/O leaves persistent memory (PMEM), made available on the system bus, out of scope for this report. In fact, PMEM is a component of storage devices or storage nodes, so we take it into consideration indirectly. It may be argued that PMEM can constitute a distinct performance layer.

- The programmer viewpoint: we survey the interfaces and abstractions available to programmers to manipulate stored data.

We conclude with an overview of the capabilities of a storage configuration that are relevant for the DAPHNE run-time.

2 Workloads

We first consider the I/O characteristics of generic workloads considered in the context of DAPHNE pipelines: HPC simulation, machine learning and high-performance data analytics. In Section 2.1, we also discuss the classes of systems designed to handle these workloads. In Section 2.2, we describe the initial pipelines designed for the DAPHNE use cases. We discuss metrics relevant for storage in Section 2.3 and the promise of computational storage in Section 2.4.

2.1 Generic Workloads

2.1.1 HPC Simulation

Large files (e.g., containing multiple 2D images, or large matrices) are read sequentially when starting a simulation and large files are written sequentially when taking checkpoints of the simulated state or storing the result of the simulation.

This is the regime for which parallel file systems (see Section 3.3.1) were designed. The large amounts of RAM in the compute nodes running the simulation have increased the pressure on parallel file systems to support efficiently large bursts of sequential writes. To deal with these bursts, specific systems based on high-performance SSDs have been introduced as an intermediate layer between compute nodes and parallel file systems. Such systems are denoted burst buffers (see Section 3.3.2).

2.1.2 Machine Learning

Machine Learning and especially Deep Learning frameworks perform large amounts of read I/Os and few write I/Os. These frameworks rely on a large number of iterations over the same data set with randomized read patterns. If all training data is loaded in memory, then this workload results in a large sequential read. Otherwise, many relatively small, random read I/Os are sent to the storage system. A storage system such as DAOS (see Section 3.3.3) was designed to handle this I/O regime.

2.1.3 High-Performance Data Analytics

HPDA applications are highly parallelized to process large (at rest or streaming) data sets efficiently. Analytics workloads often require staging of intermediate data sets and interim results for multi-step/multi-pass algorithms. This has a potential to involve massive levels of data movement. Cluster file systems, such as HDFS or Ceph, were designed to co-locate processing and data storage and thus handle such I/O regime (see Section 3.3.1).

2.2 DAPHNE Use-Cases

The initial data pipelines for the DAPHNE use cases, described in deliverable D8.1, mention stored data as follows:

- **The earth-Observation case study**: Models are trained with the So2Sat-LCZ42 data set, stored in HDF5 files. This dataset (51,8GB) is available online[3]. It must be downloaded and stored before training proceeds.

3 https://mediatum.ub.tum.de/1483140

    - The input to the pipeline are enriched satellite images (one per region of interest), stored in GeoTIFF files, while the output are LCZ maps (one per region of interest), also stored in GeoTIFF files. The proposed initial pipeline identifies load image and store to image steps as python scripts.
    - The GDAL library is proposed to write GeoTIFF files. GDAL uses its own virtual file system abstraction to access files. This abstraction enables GDAL to access in-memory files, compressed files, encrypted files, files stored on network, or cloud-based object storage services. It is built on top of the POSIX file interface (unistd.h).
- **The semiconductor manufacturing case study**: Models are trained from a CSV file obtained by combining tables from a MySQL database. Inferences are done on streaming data.
- **The material degradation case study**: A simulation is done based on sampled waveforms stored in files. The results of a reduction step are stored in a database service, which serves as input and output for the simulation.
- **The automotive vehicle development case studies**: The two types of simulations described take CSV files as input. They generate CSV or JSON files.

In summary, the initial data pipelines considered for these use cases where (i) data is stored in files (GeoTiff, HDF5, CSV, JSON, MySQL files), (ii) stored datasets are loaded entirely in memory before they are processed and (iii) the output of the pipeline is stored in files.

We expect that integrated DAPHNE pipelines will consider (a) asynchronous and parallel access to stored data, (b) modern I/O frameworks, (c) data stored on SSDs during processing, and (d) offload of pipeline fragments to computational storage for better cost-performance, better scalability, and better resource utilization.

2.3 Metrics

We mentioned that DAPHNE might leverage storage to improve cost-performance, scalability and resource utilization. Let us define these metrics:

- Cost-performance: The idea is that performance measured in terms of latency or throughput, should be balanced with the associated cost of storing data either in RAM, while it is processed, or on secondary storage, when it is not. Jim Gray defined the 5 min rule, a rule of thumb stating that data that is accessed in the next five minute should remain in RAM, while data accessed less frequently should be stored on secondary storage. Today, with SSDs and 4K pages, the threshold is closer to 1 min [1].
- Scalability: Scalability can be measured in terms of volume of datasets or complexity of the data pipelines.
- Resource utilization: Resource utilization can be measured in terms of RAM footprint or possibly energy spent processing data pipelines.

2.4 The Promise of Computational Storage

We will define the architecture view of computational storage in 3.2 and the programmer view in 4.3. For now, let us define the promise of computational storage in terms of how it may impact relevant metrics for DAPHNE workloads.

For this discussion, we simply consider that (i) computational storage makes it possible to process data where it is stored, with minimal data movement, and that (ii) computational storage can be programmed statically, or run code offloaded dynamically.

We consider that the main impact of computational storage on DAPHNE pipelines will be improved cost-performance. Executing tasks or sub-tasks (e.g., linear algebra or relational operators) involving stored data sets on computational storage will make it possible to avoid data transfers to memory and thus improve data pipeline cost-performance. For example, a machine learning pipeline might rely on quantized data. This quantization might be performed on computational storage, so that only quantized data is transferred to memory.

Computational storage will also impact scalability, as computational storage processing power can scale gracefully with the amount of stored data, as well as resource utilization, as the processors embedded on computational storage are less power hungry than the general-purpose processors of traditional compute nodes.

Existing work explores the promise of computational storage in the context of HPC [2], Machine Learning [3] or High-Performance Data Analytics [4,5,6] workloads. They illustrate the three points above: improved cost-performance due to minimized data movement, scalability and better energy efficiency.

2.5 Take-Aways

We identified generic workloads for:

- HPC simulation: large sequential reads and writes.
- Machine Learning: many small random reads and few writes.
- High-Performance Data Analytics: sequential reads and writes with staging of intermediate data sets for multi-pass algorithms.

Initial data pipelines for the DAPHNE use cases access data stored in files. These stored datasets are loaded entirely in memory before they are processed. Resulting data sets are stored in files.

Computational storage should enable better cost-performance, better scalability, and better resource utilization for DAPHNE pipelines.

3 Architecture Viewpoint

Let us review how compute and storage are combined, first within devices, then across a cluster and on the cloud.

What we usually call storage, e.g., storage devices such as disk, tape, or disk array, are not just composed of storage media. They also contain compute elements. We survey their characteristics in Section 3.1.

So, if all storage devices contain compute components, what is computational storage? We define this concept more precisely in Section 3.2.

In a HPC cluster, there may be a distinction between compute and storage nodes. We survey their characteristics in Section 3.3.

Finally, we describe the role of cloud storage in Section 3.4.

3.1 Storage Devices

We classify storage devices in two categories: *storage drives* that encapsulate storage media and *storage hubs* that expose a uniform interface to one or several underlying storage drives.

Storage drives encapsulate storage media. They are composed of three main components:

- *Host controller interface*, supporting a transport protocol (SATA, NVMe, iSCSI) over a local interconnect (SATA, PCIe) or fabric (TCP/IP, RDMA). See Appendix 1 for details on PCIe and Appendix 2 for details on NVMe.
- *Storage controller,* running firmware responsible for mapping the logical address space onto storage media. On today’s commercial drives, the storage controller is an ASIC.
- *Storage media*, where data is stored persistently (NAND flash, 3D-Xpoint, magnetic disks or tape). Multiple storage media components may be connected in parallel to the storage controller. See Appendix 3 for details on NAND flash.

Storage hubs provide a uniform interface over one or several underlying storage drives (or other hubs). They are composed of three main components:

- *Front-end interface* supporting a transport protocol (NVMe, iSCSI, REST, NFS, Lustre, RADOS) over a local interconnect (SATA, PCIe) or fabric (TCP/IP, RDMA).
- *Storage processor* running software responsible for providing storage services on top of the underlying storage drives (e.g., RAID, compression, deduplication). The storage processor is a CPU, a DPU, a FPGA or a MPSOC.
- Back-end interface connecting one or several storage drives or storage hubs. This can range from a collection of point-to-point connections to a host-based adapter connected to a backplane bus.

3.1.1 Solid-State Drives

Solid-State Drives (SSDs) are composed of tens of flash chips wired in parallel to a storage controller [7]. Today, SSDs are orders of magnitude faster than hard disks because of their intrinsic parallelism and because of the high performance of flash chips.

SSDs have efficient random access and capability to process requests in parallel. On the other hand, they do not allow in-place updates and their units wear over time. Since there are no in-place updates, there needs to be garbage collection of the older blocks. The read, write, and erase granularities of the device are, however, different.

In commodity SSDs, all this complexity is handled by the flash translation layer (FTL), which is a vendor-specific black box. The goal with this abstraction is to relieve end users from worrying about such internal characteristics of SSDs. However, being oblivious to them prevents us from exploiting the full power of SSDs. In addition, even with all this abstraction, one cannot be fully oblivious to what goes on in an SSD. For example, a write operation may trigger rewrites and garbage collection leading to write amplification and unpredictable read and write latencies for the end users.

SSDs are also not a uniform class of devices. Even the commodity SSDs that we call *generic black-box SSDs* have changing characteristics within the group. Today, in addition, there is a variety of options offering alternatives to traditional block interface and POSIX filesystem abstractions that the generic SSDs target. Some of these varieties give customization and co-design opportunities with respect to individual applications.

We refer to the SSDs that only provide the traditional HDD-compatible block interface to end-users and have a flash translation layer that is a black-box controlled only by the vendor as generic black-box SSDs. Devices under this category are the easiest to use out-of-the-box because of the compatible legacy interface. However, they are rigid when it comes to customization and co-design possibilities with the applications.

On the other hand, designing applications that are conscious of the well-known internal characteristics of SSDs still gives high benefits in terms of application performance and SSD lifetime [7]. Furthermore, based on the FTL implementation of individual SSDs, one can get different read and write latencies from request of different sizes as well as the predictability of the latency behavior would vary (e.g., due to garbage collection logic). In addition, the density of flash chips in an SSD, the number of bits stored in a flash cell, also impact performance. SLC (single-level cell), MLC (multi-level cell), TLC (triple-level cell), and QLC (quad-level cell) flash chips store 1, 2, 3, and 4 bits in a cell, respectively. The higher the density the lower the cost per gigabyte, but lower overall performance and resilience to wear. Therefore, even if it is challenging, understanding the characteristics of the different black boxes that the generic SSDs are and adapting and customizing the applications accordingly would help. Such knowledge would similarly help choosing the right type of generic SSDs for an application.

3.1.2 Hard disk drives

Hard disk drives (HDDs) have been the predominant choice for secondary storage since the 1950s. Increasingly, HDDs are replaced by SSDs in data center environments, and personal computing, due to higher data-transfer rates, higher areal storage density, and much lower latency and access times. Yet, SSDs are still about an order of magnitude more expensive (in terms of TB/$). It remains unclear whether SSDs will entirely replace HDDs for all relevant market segments, but the price trend clearly indicates that SSDs will continue to receive wider adoption compared to HDDs in the next decade.

An HDD consists of multiple platters, where each platter is organized in a set of tracks. Data is organized in blocks (sectors) of typically 512 or 4096 bytes of data. A typical HDD is equipped with two electric motors: a spindle motor that spins the disks and an actuator that positions the read/write head over the spinning disk.

Modern HDDs provide up to 20 TBs of storage capacity with data transfer rates for sequential reads/write of up to 250 MB/s and an average access latency of 2.5 – 10 ms. The read/write performance of an HDD is dominated by three aspects: seek time (time to move read/write head to correct track), rotational delay (time to rotate read/write head over sector to read), and the data transfer rate. Due to physical limitations of the moving parts of an HDD, the access latency is in the range of milliseconds. To increase the data transfer bandwidth of HDDs, recently dual-actuator HDDs were announced, which double the data transfers rates.

Traditionally, data is recorded on HDDs using Conventional magnetic recording (CMR), which leaves space between individual tracks to account for track misregistration. This impacts the areal density since portions of the platter surface cannot be fully utilized. To increase areal density of HDDs, Shingled Magnetic Recording (SMR) were introduced in the 2010s. SMR removes gaps between adjacent tracks by overlapping them (similar to shingles on a roof). Overlapping tracks are grouped into bands, denoted zones, of fixed capacity for more effective data organization and to allow for partial updates.

This shingled format of overlapping tracks has implications on the writing process to SMR drives. Data has to be written sequentially and in-place writes/updates require a rewrite of the entire band of tracks (the zone). SMR drives come in three different flavors: drive-managed (the drive itself is organizing the sequential write restriction), host-managed (the user/application is responsible for ensuring sequential writes), and hybrid (which provides a combination of drive- and host-managed). There are two different command sets, one for SCSI drives (ZBC) and one for SATA drives (ZAC), which describe the set of commands necessary to manage zones on SMR drives (i.e., REPORT ZONES, RESET ZONE WRITE POINTER, OPEN ZONE, CLOSE ZONE, and FINISH ZONE).

3.1.3 Tape

Nowadays, tape data storage is primarily used for system backup, data archive and long-term data preservation, as well as data exchange. Tape data storage fundamentally consists of two parts: a tape media cartridge and a tape drive. Typically, tape media cartridges are stored in large robotic tape libraries with ten thousands of cartridges. Through the separation of storage media and storage drive, tape provides low price in TB/$ and has a low power consumption since only tape drives consume energy. The ratio of media cartridges to tape drives varies but is typically in the range of 200:1. Additionally, tape media has a lifetime of up to 20 years (although in practice tape can wear out significantly depending on the data access frequency) and offers increased data security (e.g., to avoid ransomware attacks).

Current tape technologies offer up to 45 TB storage capacity (compressed; up to 18 TB uncompressed). The data access latency is usually in the 10s of seconds since the tape has to be moved to the correct location to read a block of data. Modern tape media is around 1 km in length and provides a maximum read/write throughput of up 1 GB/s (LTO-9) on compressed data and 400 MB/s (LTO-9) on uncompressed data. Projections for the upcoming LTO-10 standard show that the capacity will double (36 TB uncompressed, 90 TB compressed) and the read/write bandwidth will nearly triple (1.1 GB/s on uncompressed data, 2.75 GB/s on compressed data). Optional features include built-in compression and encryption that is performed directly within the tape drive.

3.1.4 Disk Arrays

Disk arrays are storage hubs making a collection of storage drives available to multiple hosts on a network. There is a traditional distinction between disk arrays providing a file abstraction, denoted Network-Attached Storage (NAS) and disk arrays providing a block abstraction, denoted Storage-Area Network (SAN). NAS and SAN used to be based on Hard Disk Drives. They are now based on SSDs, and denoted All-Flash Arrays. Recently, vendors have commercialized functional accelerator cards that are basically a disk array front-end without storage drives.

Disk arrays that expose a block abstraction expose a collection of volumes. Volumes may reflect the underlying storage drives, as JBOD (just a bunch of drives), or bundle several storage drives and leverage parallelism and redundancy to provide high performance and availability (RAID).

The storage processor of the disk arrays provides storage services, such as deduplication, compression, or encryption. They leverage redundancy across network cards (multipath access) to provide high availability.

3.2 Computational Storage

Computational storage denotes the integration of programmable compute resources within storage devices. Computational storage makes it possible to run portions of data-intensive application on storage devices. It promises reduced data movement, better energy efficiency and reduced costs. Indeed, the compute resources introduced within storage devices are specialized processing units, denoted data processing units (DPU) equipped with hardware accelerators that are cheaper and more energy efficient than the general-purpose CPUs.

3.2.1 Background

The idea of offloading processing to storage was first expressed twenty years ago. Pioneering efforts explored the design of *active disks*, i.e, hard disks drives equipped with programmable storage controllers [8]. At the time however, there was no need for offload as advances in silicon technologies led to exponential improvements in CPU performance. Yet, Jim Gray expressed confidence that active disks would become relevant in the future. In his presentation titled “Put Everything in Future Disk Controllers (it’s not if, it’s when”) [9], he argued that running application code on disk controllers would be (a) possible because disks would be equipped with powerful processors and connected to networks via high-level protocols, and (b) necessary to minimize data movement. He concluded that there would be a need for a programming environment for disk controllers.

3.2.2 Architecture

Both storage drives and storage hubs contain compute components in the form of a storage controller or processor. If that component is programmable, then we have a computational storage device:

1. Computational storage drives are storage drives equipped with a programmable controller. For example, the Cosmos+ and Daisy+ OpenSSD prototypes are equipped with MPSoC (multiprocessor systems that combine CPU cores and FPGA).
2. Computational storage hubs are storage hubs whose storage processor is programmable. For example, the Daisy OpenSSD prototype is based on MPSoC connected to 2 x M.2 SSDs as well PCIe and 100G Ethernet connectivity.

There is a distinction in the literature between (a) near-storage or near-data processing [10], that corresponds to programmable storage hubs and (b) in-storage processing that corresponds to programmable storage drives.

A task force at SNIA, a trade group representing storage companies, has defined terminology and architectures for computational storage [11]. They denote the processing units integrated with storage as *computational storage processors*. When combined with traditional storage drives or hubs, they provide Computational Storage Services (CSS) to hosts.

3.2.3 Computational Storage Services

SNIA distinguishes between fixed computational storage services, based on predefined functions such as compression or encryption, and programmable computational storage services. In the remainder of this document, we only consider programmable computational storage[4], that we refer to as computational storage for the sake of simplicity.

Programmable computational storage services are installed via code upload. SNIA lists four code upload mechanisms:

1. operating system image, installed on a CPU.
2. containerized application, installed on a a container engine.
3. FPGA bitstream, installed on a FPGA.
4. eBPF bytecode (eBPF is a vendor-neutral Instruction Set Architecture already used for offloading code to Network Interface Cards), installed on an eBPF virtual machine.

For instance, the Daisy platform, with is MPSOC, can accommodate these four types of code uploads, if it runs a container engine and an eBPF virtual machine.

An extension of the NVMe standard for computational storage is expected in 2022. Such a standard will define mechanisms for shipping eBPF code to computational storage. SNIA already defined principles for such an API, based on primitives for administration (inspection, memory allocation and association to storage space) and code shipping.

In the context of DAPHNE, we focus on the definition of predefined programs (operating system image or containerized applications), hardware acceleration (FPGA) and code shipping (eBPF) that improve the performance and scalability of integrated data pipelines.

4 As a result, we do not cover devices that provide fixed computational services, e.g., ScaleFlux which incorporates a fixed compression service.

3.2.4 Computational Storage Devices

Computational storage is a very active area of research. Surveys have been conducted to review the state-of-the-art and identify open issues in the domain [10], [12]. The most recent survey of computational storage [12] identifies two platforms openly available for research: DFC and OpenSSD. Both are storage hubs, equipped with both ARM processor and FPGA. DFC is now discontinued. We mentioned above the Cosmos+, Daisy and Daisy+ OpenSSD prototypes. Existing commercial devices include NGD (storage hub equipped with an ARM processor targeting the offload of containerized applications) and Samsung SmartSSD (storage hub with FPGA).

3.2.4.1 DAISY

Daisy is the latest iteration of OpenSSD prototypes, designed by Prof. Song and his group at Hanyang University[5]. Daisy is a storage hub equipped with 2x100GE and PCIe Gen3x16 connectors, a Zynq Ultrascale+ MPSoC and a backplane interface for connecting to two M.2 SSDs.

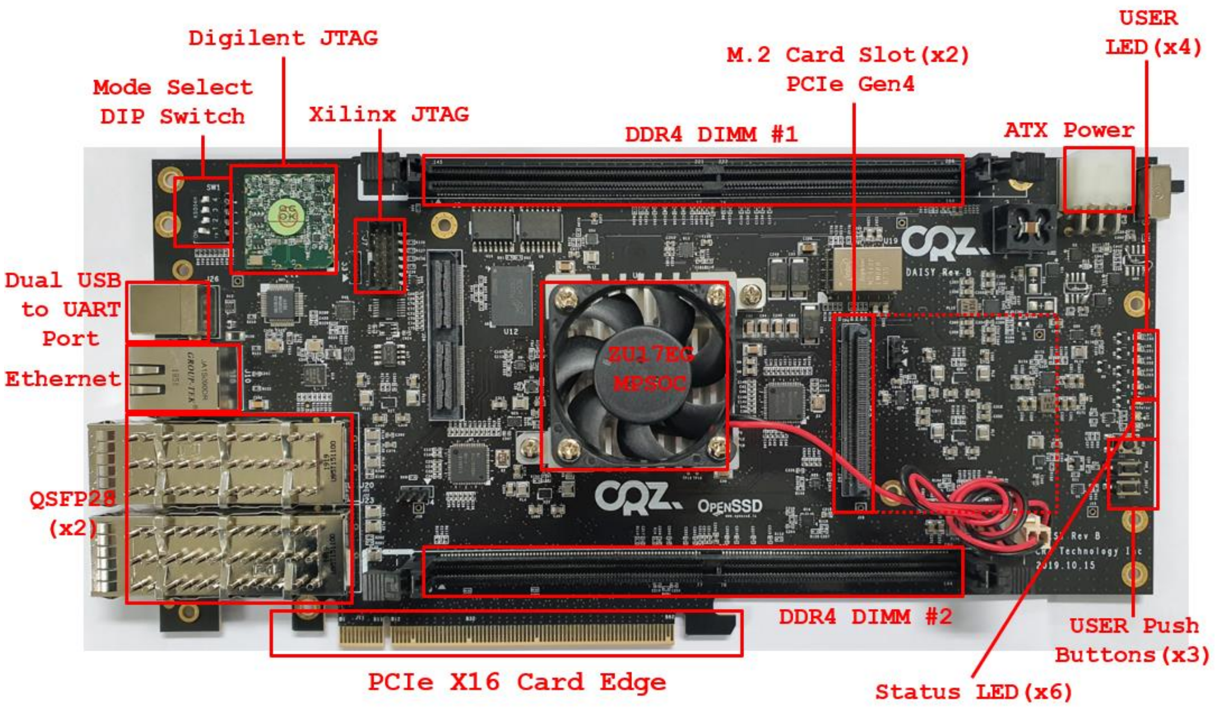


Figure 1: OpenSSD Daisy Architecture

The Zynq Ultrascale+ MPSoC is a heterogeneous multiprocessing platform for embedded applications. Such a MPSoC combines hardware acceleration on FPGA with the flexibility of ARM cores running embedded linux (e.g., SSD interface, bytecode execution). This platform can support the four types of code uploads associated with programmable computational storage services according to SNIA:

5 http://openssd.io/

- Xilinx IP can be used for generic hardware acceleration components, e.g., NVMe controller or DMA engines over PCIe in the context of a storage processor, while specific accelerators can be written with a hardware description language and synthesized into a bitstream. A bitstream including generic and/or specific IP can be included into the Daisy boot package.
- The software run on the storage processor is cross compiled for the ARM cores, and also packaged in the Daisy boot package.
- Since 2019, the docker container engine runs with embedded linux on ARM cores. The RAM available on the Daisy should be enough to run the container engine, as a permanent service on the storage processor, and thus make it possible to upload containers at run-time.
- We are building a software component that will support eBPF upload on the Daisy. See details in Section 4.3.2.

Daisy+ is a computational storage drive under development, similar to Daisy, where the MPSoC is also connected to a NAND flash So-DIMM component.

3.2.4.2 Intel Kestral

Kestral is a programmable storage drive, combining a PCIe Gen4 x16 connector, a Stratix 10 DX FPGA, including an integrated quad-core 64-bit Arm* Cortex*-A53 hard processor subsystem, and 2 TB of Optane persistent memory. A MAX 10-based system controller is used to sequence and monitor power supplies for the Stratix 10 DX FPGA.

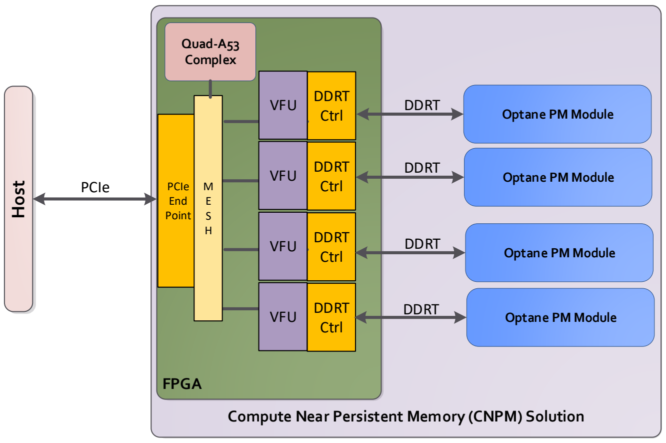


Figure 2: Intel Kestral Architecture

Such a device can be used as an ultra-low latency computational storage drive, standalone computational memory accelerator, or as an P2P compute device with large memory.

3.2.4.3 Bittware IA-840F

The Bittware IA-840F is a programable storage hub, combining a PCIe Gen4 connector equipped with a SMBus controller, and an Agilex AGF014 FPGA.

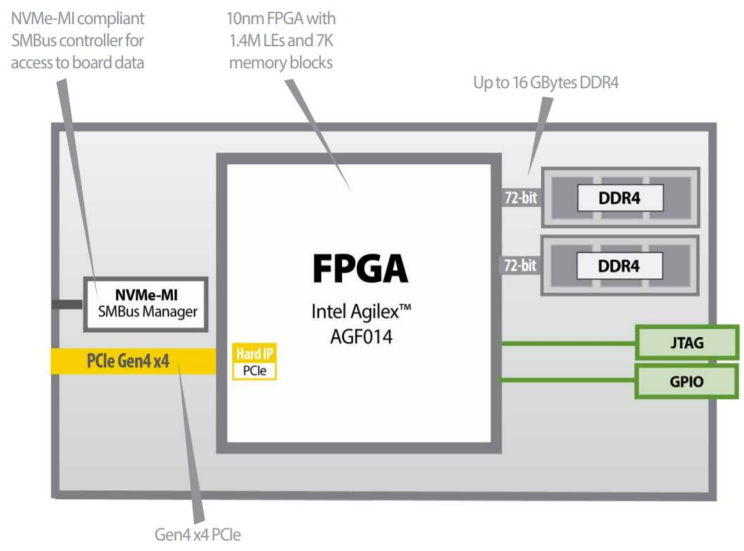


Figure 3: Bittware IA-840F Architecture

This storage hub has no direct attached storage (unlike Daisy) and is intended to be deployed within conventional U.2 NVMe storage array. It acts as a computational storage processor for data stored on NVMe SSDs connected to the same PCIe root complex (see Figure 4 below).

3.2.4.4 Bittware IA-220-U2

The Bittware IA-220-U2 is a programmable storage hub, combining interconnect (PCIe Gen4 x16) and fabric connectivity (3x200GbE), with an Agilex FPGA and MCIO expansion ports to connect PCIe SSDs (4 x4 PCIe SSDs, 2 x8 PCIe SSDs, 1 x16 PCIe SSD). A BMC processor is available for board management and control, as well as debugging. This platform is targeting bitstream offload, and thus hardware accelerated processing of stored data.

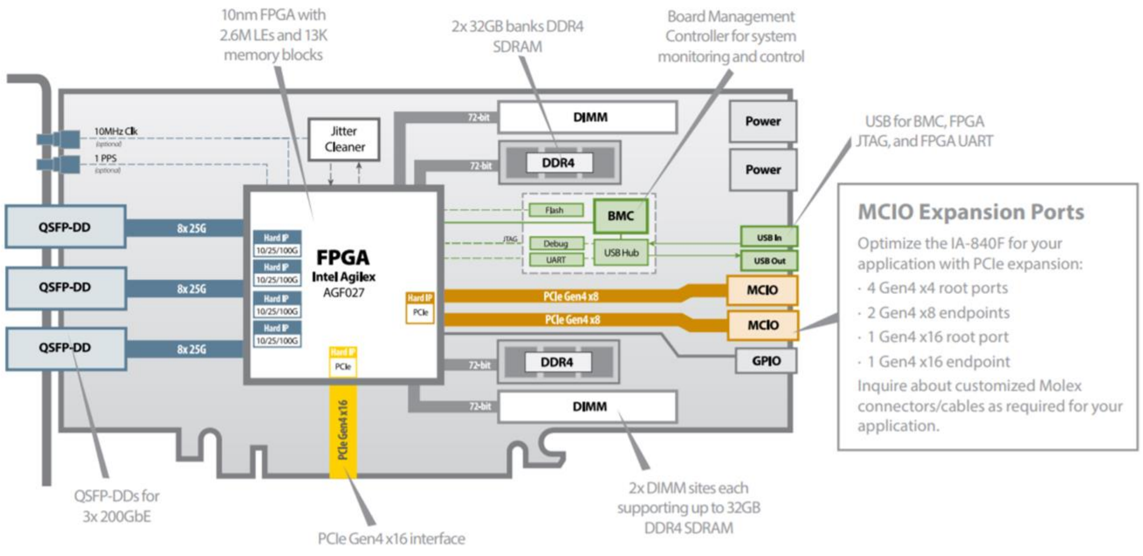


Figure 4: Bittware IA-220-U2 Architecture

Figure 4 illustrates the IA-220-U2 used as a computational storage processor for NVMe SSDs connected to the same PCIe root complex. Data movement between NVMe SSDs and the computational storage processor hub is orchestrated through peer-to-peer DMA. Eideticom's libnoload library is available on the host to orchestrate exchanges across disks and computational storage processor. The FPGA runs Eideticom's NoLoad IP as a storage processor.

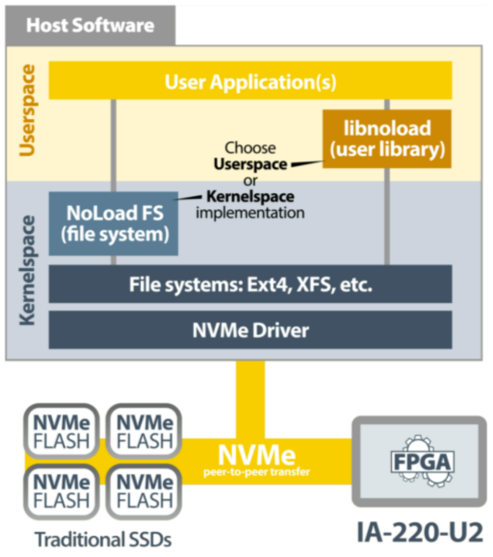


Figure 5: Host/Bittware IA-220-U2/SSD architecture

3.3 Cluster Storage

Ideally, public cloud or HPC cluster resources can be provisioned to match the changing needs of applications. To support independent scaling, computation and storage are isolated on different nodes. Storage is thus accessed over a network rather than locally. Compute nodes are equipped with a central processing unit, possibly accelerators (e.g., GPU, TPU, FPGA). They run the Machine Learning platforms or HPDA engines that underlie data science pipelines. Data is obtained from storage nodes, equipped with a processor and a collection of disks.

The role of the compute nodes is to perform computations. They are independent machines running their own operating system on hardware such as CPU/GPU, RAM, a network interface controller (NIC), and possibly storage drives, also denoted as on-node storage, for storing temporary data. The compute nodes are grouped together in a cluster that allows them to work together on demanding and complex tasks.

The role of storage nodes is to service the needs of compute nodes in terms of access to stored data. The storage nodes are independent computers with NIC, CPU, and attached to storage drives or storage hubs. Storage nodes are typically arranged into a distributed storage system. Storage nodes can be used for performance, capacity or archival. The storage hardware used for performance are NVMe SSDs, while HDDs or SSDs are used for capacity and tapes are used for archival.

On HPC clusters, the fabric between compute nodes is low latency and high throughput. RDMA is typically used for this purpose. It allows for data transfer between nodes, without involving the processor, cache, or operating system of either node, making data transfer much more efficient than the traditional TCP/IP stack. RDMA is also the storage fabric of choice for high-performance storage devices. It supports the fastest implementation of NVMe over fabric (NVMeOF), which is a network protocol for direct communication with NVMe drives. NVMeOF can handle the same high queue depths as NVMe drives which allows for a highly parallel architecture where compute nodes are able to fully saturate the bandwidth of the IO nodes. For slower storage devices, the requirements on the fabric are less stringent. As a result, traditional TCP/IP stacks are used.

3.3.1 Parallel File Systems

A parallel file system is a distributed system, where a collection of dedicated server nodes (attached to storage drives or storage hubs) store portions of files, denoted stripes, and enable multiple client nodes to access them in parallel [13]. Parallel file systems provide (i) a global name service, mapping files onto stripes (ii) a locking service to enforce consistency guarantees in the presence of concurrent accesses. Parallel file systems provide location and concurrency transparency: clients are not aware of where files are located, and they share the same view of the state of the file system.

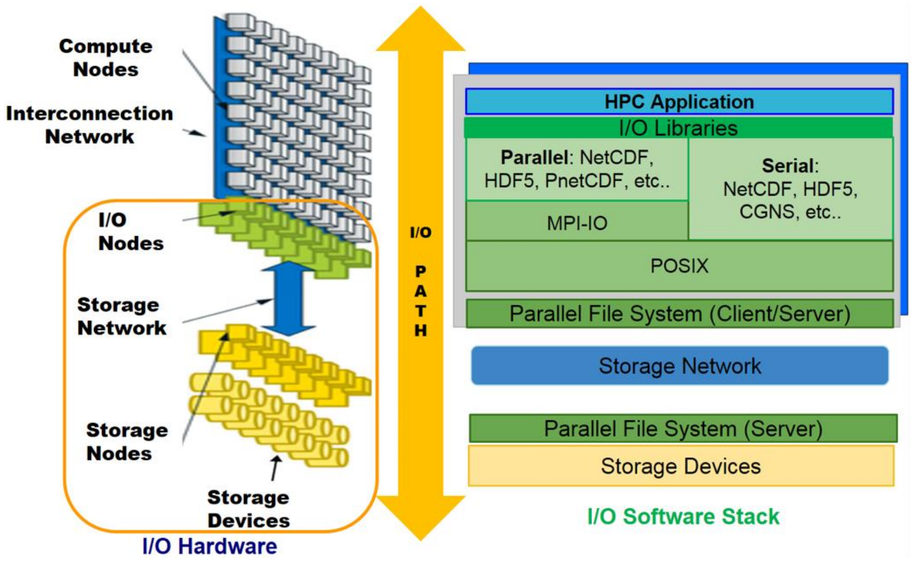


Figure 4: Parallel File System (Architecture and Programmer views).

Parallel file system clients coordinate parallel accesses to a single file, based on one of the following patterns:

- Single writer: all clients communicate and let one of them perform I/Os.
- Multiple writers: the file is partitioned, and each client performs I/O on a given partition.
- Collective writers: A subset of tasks is dedicated to performing I/Os. They collect I/O requests from compute tasks and coordinate with each other as multiple writers.

GPFS is a parallel file system emulating closely the behavior of a general-purpose POSIX system running on a single system. A file consists of blocks of equal size, ranging from 16 KB to 1 MB striped across several nodes. GPFS consistency and synchronization are ensured by a distributed locking mechanism. A central lock manager grants lock tokens to local lock managers running in each server node. GPFS has a POSIX compliant API. It can be mounted on a compute node using the virtual file system interface in Linux.

A Lustre file system is composed of one or more metadata servers. Unlike GPFS, the meta-data server is not involved in data transfers, just in permission checks. one or more object storage servers store file data. Clients have concurrent and coherent read and write access to the files in the filesystem via a unified namespace for all files, with standard POSIX semantics.

We refer interested readers to Prace's Best Practice Guide for Parallel I/O for more details on GPFS and Lustre [13]. As indicated in Figure 6 (stemming from [13]), access to a parallel file system might be encapsulated within a programming language library (e.g., that expose persistent data structures).

Cluster-file system such as HDFS or Ceph share many of the characteristics of parallel file systems: striping, global name service, locking. The big difference between these two classes of systems is that cluster-file systems break location transparency to enable co-location of processing and storage. HDFS and Ceph maintain rack-awareness meta-data and can inform a cluster resource manager (e.g., YARN or Kubernetes) of the file stripes located on a given node.

3.3.2 Burst Buffers

Burst buffers are composed of dedicated storage resources, either on compute nodes, or on intermediate nodes (denoted IO nodes). Their purpose is to quickly absorb bursts of writes from the compute nodes, allowing the computation to resume while the IO nodes are offloading the data to the (slower) storage nodes.

IO nodes can also be used to multiplex IO by shaping many small writes into larger sequential writes that are more easily digestible by the PFS. Another performance optimization is to use the IO nodes to stage data that will be needed by the compute nodes, allowing faster access and random-access performance. All these scenarios reduce the time that compute nodes spend on I/Os.

3.3.3 DAOS: Distributed Asynchronous Object Store

DAOS is a distributed storage system with a client-server model [14]. The DAOS client is a library integrated with applications running on the compute nodes. The library exposes stored data directly through the DAOS API or through bespoke interfaces built on top of the DAOS API.

The DAOS server, denoted storage engine, is running on the storage nodes and is responsible for managing access to PMEM and NVMe SSDs. PMEM is used to store internal metadata, file metadata and I/Os smaller than 4KB. NVMe SSDs are used to store application data. PMEM is accessed via the Persistent Memory Development Kit (PMDK), while the Storage Performance Development Kit (SPDK) is used for NVMe SSDs. Note that the DAOS storage engine does not use SSDs as raw devices. Instead, it relies on the file abstraction (blobstore) available in SPDK.

DAOS clients and servers are connected via RDMA-enabled fabric. They communicate with the Mercury RPC library running atop libfabric, a library exposing the underlying high-throughput/low-latency fabric to user-space[6].

The DAOS storage model is composed of node, pool, target, container and object. A node is a physical component that consists of one or two DAOS storage engines.

6 Libfabric is the core component of the Open Fabrics Interface (OFI).

A pool is a reservation of PMEM and NVMe storage on given DAOS storage engine. A target is a collection of pool shards distributed across multiple nodes. The pool keeps track of its associated targets in a versioned list called a pool map. Besides keeping track of the active targets it also stores a storage topology in a tree structure which is layered to identify targets in the same fault domains i.e. running on the same hardware.

A container represents an object address space inside a pool. It is reachable by its unique identifier. An application can open the container by connecting to the proper pool and requesting access.

An object is the representation of a dataset. There are two different types of objects: (a) array objects which are one-dimensional arrays of fixed size elements, and (b) key/value sets supporting the traditional put, get, remove and list operations. For example, a single HDF5 object can be represented as a key-value set.

3.4 Cloud Storage

This report focuses on HPC clusters. We should note that cloud services can be used to run integrated data pipelines and that HPC clusters might rely on cloud storage services (e.g., for archival).

Here is thus a brief overview of cloud storage [15]–[18]. A compute instance on the cloud can access various forms of storage:

- Local SSDs in the case of bare metal instance;
- Block storage, exposed through a storage hub, and accessed exclusively via I/Os (e.g., AWS EBS);
- Storage services, accessed via a network: e.g., object storage available through a REST API (e.g., AWS S3 or Glacier), file sharing service (e.g., AWS EFS), or database service (e.g., AWS RDS, DynamoDB or Redshift).

3.5 Take-Aways

We identified various forms of storage systems available in a HPC system:

- Storage devices that can be directly accessed from compute nodes. We distinguish storage drives and storage hubs, depending on whether the devices include storage media or not.
- Parallel file systems, burst buffers and object storage systems supported by dedicated storage nodes in the HPC cluster (e.g., GPFS, Lustre, Ceph) or installed on compute nodes (e.g., HDFS).

We detailed the characteristics of programmable computational storage services, installed via code upload on computational storage devices. We described four types of computational storage devices.

4 Programmer Viewpoint

From a programmer viewpoint, storage systems are exposed through various interfaces. Figure 7 illustrates a set of solutions that may be available on a Linux cluster. Applications can access I/O devices, directly via a file system, or a distributed storage system.

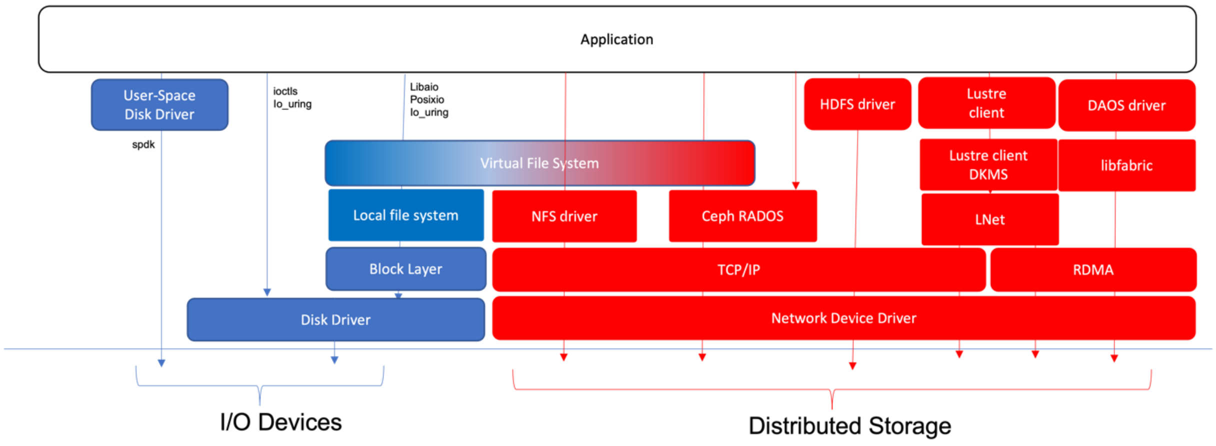


Figure 5: Example storage systems in a HPC cluster

In the rest of this section, we review the multiple I/O frameworks in Section 4.1 and abstractions that applications can use to access stored data in Section 4.2. We discuss how to program computational storage in Section 4.3.

4.1 I/O Frameworks

4.1.1 POSIX I/O

POSIX I/O is a simple and beautiful abstraction based on files as arrays of bytes. The interface makes it possible to create/delete, open/close and read/write from a file. The traditional POSIX I/O relies on synchronous system calls accessing a file system through the C standard library (unistd.h).

The limitations of the POSIX API in the context of HPC are well documented[7]: parallel accesses challenge both the array of bytes abstraction and the strong consistency guarantees associated to reads and writes. More generally, it has been shown that building crash-consistent systems atop the POSIX API is very challenging because there is no standard definition of how failures impact I/Os [19].

4.1.2 AIO, LIBAIO

Since the mid-2000s, there has been two flavours of asynchronous I/Os in Linux: aio and libaio. The former, aio, is a user-space emulation of POSIX asynchronous I/Os that relies on worker threads issuing synchronous I/O calls. It provides functions for submitting I/O requests (e.g., aio_read, aio_write, aio_fsync). The asynchronous I/O control block (aiocb) is passed as a parameter to these functions. This control block defines (i) the target file and offset, (ii) a pointer to the data buffer (from which data is read, or where data is written), and (iii) a

7 http://www.pdl.cmu.edu/posix/docs/POSIX-extensions-goals.pdf

notification method (i.e., a signal or a callback function). As aio relies on synchronous I/Os, it works on any file system, with buffered or unbuffered I/Os.

The latter, libaio, issues asynchronous I/Os through the io_submit system call. This system call takes as input a context (initialized through the io_set up system call), and a control block (iocb) that contains the same attributes as those described above for aiocb, together with a code describing the I/O being performed (read or write). On completion, the system call io_getevent is used to reap io_event data structures that may contain a pointer to a callback function. Note that libaio requires that a file be opened with O_DIRECT. It only works with the following file systems: ext2, ext3, jfs and xfs. Libaio requires two system calls per I/O.

4.1.3 IOCTL

Once a device file is opened, I/Os can be submitted on the raw device through one of the I/O interfaces presented above, or directly through ioctl requests. Passthrough device access via ioctl enables a direct access from user-space to all functionalities exposed by the NVMe driver. There is a caveat though. The ioctl system call is synchronous.

4.1.4 IO_URING

io_uring was introduced by Jens Axboe in 2019 [20]. It is a flexible and efficient mechanisms to manage I/Os. It relies on pairs of circular submission/completion queues shared between user-space and kernel-space to minimize software overhead. The queues are single producer and single consumer. Submission queue entries are 64B, while completion queue entries are 16B. The interaction between user-space and kernel-space through queue pairs mirrors the interaction between host driver and NVMe controller.

System calls are available for (a) setting up queues, (b) registering application memory referenced in submission queue entries and (c) initiating/completing a number of asynchronous I/Os. By default, io_uring requires a single system call for multiple I/O submissions and completions.

It is also possible to setup a queue pair with a flag (IORING_SETUP_SQPOLL) so that io_uring starts a kernel thread that polls the shared submission queue for entries. This polling mode on the submission queue trades increased CPU utilization for reduced I/O latency.

io_uring is available through liburing that defines a range of helper functions for setting up queues, manipulating buffers and generating submission queue entries. io_uring can be used with any file system with buffered or unbuffered I/Os. According to Jens Axboe, its throughput is 2x (default) or 3x (with submission polling) compared to libaio on a single core [20]. The downside is that io_uring introduces subtle dependencies across opcodes used in submission queue entries, buffer or file registration and polling modes [21].

4.1.5 SPDK

In 2015, Intel introduced the Storage Performance Development Kit (SPDK), that bypasses the Linux kernel to access SSDs[8]. It is based on a user-space NVMe driver, packaged as a library that maps PCIe BARs directly into the application process thus supporting zero-copy. Functions

8 https://spdk.io

are provided to allocate the queue pairs used for I/O submission and completion, as well as payload buffers in this DMA-transferrable memory. NVMe SSDs can be accessed directly via a C API (nvme.h for block devices nvme_zns.h for ZNS drives). The application must ensure that a single thread submits I/O and that this thread polls for completion. SPDK thus trades increased CPU usage for performance.

In addition, Intel's Open Storage Toolkit has support for computational storage. Though not yet open source, the software is available by request. It includes both host and target-mode code for prototyping complete end-to-end solutions [22].

4.1.6 xNVMe

POSIX I/O used to be the only option for accessing stored data, until the mid-2000s. The introduction of asynchronous I/Os resulted in a multiplication of POSIX compliant I/O interfaces. The introduction of NVMe SSDs has compounded the problem. The fact that multiple storage interfaces are available for data-intensive systems is a problem for system design. Indeed, choosing a single I/O interface is a "difficult-to-reverse" decision with major implications on system design, and supporting multiple I/O interfaces is expensive.

xNVMe is a user-space library that provides a uniform API for device management, memory management and I/O submission over existing I/O interfaces[9]. It was introduced by Samsung in 2020. It provides programmers with a simple and extensible framework at negligible performance penalty [21].

4.2 Storage Abstractions

4.2.1 Blocks

The block device abstraction exposes storage as an array of fixed-sized blocks. Each block is an array of bytes. It is equipped with a logical block address (LBA). Read and write operations are defined on the flat LBA address space.

Logical blocks are mapped onto storage media by the storage controller/processor.

4.2.2 Key Values

SNIA standardized a key-value storage interface. In NVMe, the idea is to introduce Key Spaces, i.e., NVMe namespaces of type Key-Value. The interface defines commands to manipulate Key Spaces and to store, retrieve or delete key-value pairs (asynchronously). The size of a Key Space is defined when it is created. It is possible to associate minimum and maximum key/value sizes for a given Key Space.

4.2.3 Zones

A disk can be exposed as a collection of zones. Each zone is an array of logical blocks, which are the unit of read and writes. Logical blocks must be written sequentially within a zone. A zone must be reset before it is written again. This is a form of append-only storage abstraction that shields the host from the complexities of the physical address space. Zones align with the concepts of Shingled Magnetic Recording (SMR) disks, which provide track zones.

9 https://xnvme.io

Zones are provided by a specific kind of NVMe drives (ZNS), SATA (ZBC) or SAS/SCSI (ZBC) drives. ZNS accommodates the disk models defined for ZAC/ZBC: (i) Host managed and (ii) Host aware. In host managed mode, applications use explicit zone transitions as well as a write command andare responsible for writing logical blocks sequentially within zones. In host aware mode, applications use implicit zone transitions, but are still responsible for maintaining a write pointer within a zone. From the point of view of applications, ZNS drives can be accessed through a POSIX interface (files or block I/Os) or through a specific interface (ZBD) that exposes zones to the application.

ZNS departs from ZAC/ZBC in several ways. The main addition is the append command, which allows applications to submit a collection of writes asynchronously on the same zone at a large queue depth. It is then the SSD controller's responsibility to guarantee that appends result in sequential writes within a zone. Append is thus a form of nameless write [10]. The other features incorporated into ZNS focus on the need for a high performant hardware interface, which departs from the design objectives of ZAC / ZBC. Examples of these features include a copy command to offload the host-managed garbage collection or a random write area that provides the host with a window to perform updates, which reflects the typical operation of a filesystem updating its metadata. Since the host model is similar to ZAC/ZBC, ZNS drives are able to leverage existing zoned ecosystems present in current operating systems. In Linux, drives can be accessed through a POSIX interface (files or block I/Os) that hides zones or through a specific interface (ZBD) that exposes zones to the application.

4.2.4 Open-Channel

The Open-Channel Interface, defined in the context of LightNVM [23], exposes SSD internals and requires a host-based FTL to manage data placement and I/O scheduling. Generic FTLs are defined in the Linux kernel, and now also in SPDK, to provide a block device abstraction on top of Open-Channel. Application-specific FTLs can also be developed on top of Open-Channel, e.g., using the OX framework [24]. The address space is organized hierarchically in groups/parallel units/chunks that reflect SSD internals. SSD characteristics impact the nature of the address space. In particular, the unit of write is a logical block, composed of one or several sectors, depending on the SSD (e.g., 24 sectors on a dual-plane TLC drive, corresponding to 4 (sectors per page) * 3 (paired pages) * 2 (planes)). The Open-Channel interface is directly accessible from user-space through xNVMe. It defines administration commands (to access the device geometry) and scatter-gather reads and writes. A chunk reset command guarantees that a chunk is erased before it is written again. Finally, the chunk copy command supports the copy of logical blocks within the Open-Channel SSD.

4.2.5 Files

In Linux, file systems implement the Virtual File System (VFS) abstraction. VFS defines data structures (superblock, inode, dentry, file) and system calls that are common to all file systems (mount/unmount, open/close, read/write).

Local file systems such as ext4, xfs, btrfs or f2fs implement different layout and allocation policies. File systems such as f2fs and zonefs provide native support for ZNS SSDs [8, 16]. Network file systems, such as NFS or CephFS also implement the VFS abstraction. As a result, the file abstraction makes it possible for applications to seamlessly access locally or remotely stored data.

With file systems, reads and writes are buffered by default. The O_DIRECT flag makes it possible to bypass the file system cache and transfer data directly between application and SSD. If buffered I/Os are used, the fsync system call flushes all data and meta-data associated to a file onto disk. When opening a device file, the SSD is accessible as a raw device. Read and write operations at a given file offset are then passed on from VFS to the block layer without interference from a file system.

The block layer receives *bio* requests. It may reorganize them, independently on each core, in software queues. Bio requests are then dispatched to the NVMe driver via hardware queues (submission/completion). This multi-queue design was introduced in 2013 to avoid lock thrashing on multicore CPUs and to accomodate NVMe SSDs [25]. The multiqueue block layer now incorporates write ordering control for ZNS SSDs via the mq-deadline block I/O scheduler.

The block layer provides various mechanisms to deal with completion: signal-based, continuous polling or hybrid polling. These mechanisms offer different trade-offs between latency and CPU utilization.

Logical filesystems require data and metadata to be stored on the data storage medium. This is an issue for tapes, which favour sequential accesses. As a result, storing metadata in one place and data in another requires many slow repositioning actions on most tape systems. Oftentimes only a trivial filesystem is supported in which files are addressed by number. In contrast, the Linear Tape File System (LTFS) is a method of storing file metadata on a separate part of the tape (using so-called tape partitions). This makes it possible to copy and paste files or directories to a tape as if it were just like another disk but does not change the fundamental sequential access nature of tape.

4.3 Programming Computational Storage

With computational storage, it is possible to program the storage infrastructure so that it meets the needs of data-intensive systems and applications. What does “programming the storage infrastructure” actually mean? According to Do. et al [12], this entails:

1. *Defining new storage interfaces*: Programmable computational storage can expose storage with a variety of interfaces, well suited for a given purpose, e.g., ML preprocessing pipeline, data analysis pipeline, transactional object store, database storage system or key-value store. Possibly, the same stored data can be exposed with different interfaces, depending on the software deployed on computational storage.
2. *Shipping code from host to storage*: DAPHNE, as other modern database systems and declarative machine learning platforms, compiles queries into executable code. Portions of data pipelines (i.e., fused operators) could be compiled into executable bytecode that is shipped to computational storage to improve performance and reduce costs.

Programming the storage infrastructure is challenging because performance and security requirements are strict, debugging is difficult, and errors can result in corrupted data or unusable devices. In this section, we review existing work related to programming computational storage and we discuss how the code upload capabilities identified by SNIA can be leveraged for programming computational storage.

4.3.1 Programming Frameworks

Computational storage solutions have been deployed for a given data-intensive system on a specific hardware platform, both on public clouds (Alibaba [5] and AWS [26]) and HPC clusters (Los Alamos National Lab [27]). Likewise, most research prototypes focus on a single data-intensive application: a database system [28], a dataflow processing engine [4], a machine learning platform [3], similarity search [29] or scientific computing [2].

A few projects have proposed software frameworks for computational storage, most notably OX [1], Willow [30], Biscuit [31] and INSIDER [32]. OX proposes an abstract execution model for defining new SSD interfaces. The OX model focuses on mapping commands defined on an application-specific address space (e.g., updating a batch of variable-sized objects) onto standard storage commands on a physical address space (e.g., writing data blocks at given addresses). The OX model defines a set of components sandwiched between an NVMe controller exposed to the host, and an NVMe driver connected to one or several Open-Channel SSDs. Using the OX model, a new storage interface was tailored to fit the needs of the BW-Tree, a database storage manager developed at Microsoft Research.

Willow considers that SSD apps are deployed on computational storage processors running a custom operating system. RPC is used to communicate between host and SSD app. SSD apps cannot be composed. Stored data is accessed via a local file system.

Biscuit is a run-time system embedded on computational storage that supports pipelines of tasks. Tasks are programmed in C++, compiled on the host and shipped to computational storage. Tasks rely on a local file system to access stored data. Biscuit is implemented directly atop a storage controller, within a proprietary Samsung SSD prototype. The potential benefits of Biscuit are illustrated with pointer chasing, string search and a database filter.

INSIDER relies on a FPGA-based reconfigurable controller as computational storage processor. INSIDER provides a file system abstraction on the host. Operations on computational storage are organized as a pipeline of sub-programs. They rely on a customized I/O stack to access stored data.

4.3.2 Leveraging SNIA Code Upload Capabilities

As described in Section 3.2.2, SNIA identifies 4 types of code upload mechanisms: OS image, container, FPGA bitstream and eBPF bytecode shipping. We discussed how various cards support these mechanisms.

Before we discuss how these mechanisms can be used for various purposes, let us remind ourselves that computational storage devices are (networked) embedded systems. They are deployed for a purpose with no interactive user interactions. As a result, we can model computational storages as running an event loop that dispatches incoming messages (from fabric or interconnect) and processes them based on pre-installed software components. In this context, we can see the various purposes associated to the four types of code upload:

1. OS image and container upload correspond to a static and a dynamic definition of the pre-installed software components (typically in the context of embedded linux).
2. FPGA bitstream corresponds to the definition of hardware-accelerated functions.
3. eBPF bytecode corresponds to the definition of functions that can define new event loop workflows, based on pre-installed software (OS, container) and hardware-accelerated (FPGA bitstream) components. The goal with eBPF byteload is not to replace OS/container/FPGA images, but to leverage their capabilities in various ways based on run-time decisions on the host.

In the rest of this section, we discuss code shipping in more details.

4.3.3 Code Shipping

Let us first focus on the mechanisms of code shipping. We consider that bytecode is generated on the host and loaded onto computational storage where it is interpreted or JIT-compiled.

We base our discussion on eBPF bytecode, as it is mentioned by SNIA. Alternatives should be considered for code shipping on computational storage, including web assembly (see [33] for a comparison of eBPF and web assembly). Note also that recent work is considering eBPF for reducing storage latency within the Linux kernel [34]. While related, this previous work does not address how eBPF is relevant for computational storage.

4.3.3.1 eBPF Context

The Berkeley Packet Filter (BPF) introduced, in 1992, the possibility to run functions from user space within the Linux kernel. BPF defined a bytecode structure together with a virtual machine running within the Linux kernel. Alexei Starovoitov introduced eBPF in 2014, as an adaptation of BPF for modern processors. There is no standardization body for eBPF. Until one is established, the latest version of the eBPF bytecode is the one which can be interpreted with the virtual machine and JIT compilers of the latest Linux kernel.

In-kernel eBPF JIT compilers are defined for x86_64, arm64, ppc64, s390x, mips64, sparc64 and arm architectures. Most eBPF instructions can be mapped directly onto native instructions of the underlying architecture.

A user-space virtual machine, uBPF, was defined in the context of the IOVisor project. uBPF is available on github[10]. It is an Apache-licensed library for executing eBPF programs, as opposed to the Linux kernel implementation which is under GPL license.

Both gcc and clang/LLVM have eBPF backend, i.e., they can generate eBPF code from C programs. Interestingly, LLVM introduces an eBPF assembly language, which makes eBPF byte code human-readable and an obvious target for code generation.

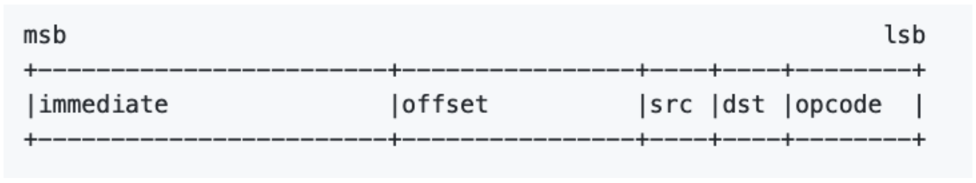
In the context of computational storage, eBPF bytecode is shipped from the host to the computational storage device that should load the bytecode onto the virtual machine (or JIT-compile the bytecode) and execute it. The goal is to run bytecode, not to run functions from

10 https://github.com/iovisor/ubpf

user space within the kernel. As a result, we focus the rest of our discussion on the uBPF virtual machine, running in user-space.

4.3.3.2 eBPF instructions

An eBPF program is a sequence of 64-bit encoded instructions (see Figure 8).



```
msb                                                        lsb
+------------------------+----------------+----+----+--------+
|immediate               |offset          |src |dst |opcode  |
+------------------------+----------------+----+----+--------+
```

Figure 6: Structure of eBPF instructions

Every instruction is encoded onto 64 bits. It is composed of:

- 8 bit opcode. Opcodes are grouped into "instruction class" based on the low 3 bits of the opcode. These classes include load, store, ALU, byteswap and branch instructions[11]. The operation can be based on registers (source and destination) or immediate operands. The load and store operations can manipulate data on the stack at 32/16/8 bits granularity.
- 4 bit destination register (dst)
- 4 bit source register (src)
- 16 bit offset
- 32 bit immediate (imm)

The call instruction uses imm as the index in the function pointer table. The first five registers hold the parameters of the external function.

In summary, eBPF can be seen as a platform-independent Instruction Set Architecture.

4.3.3.3 uBPF Virtual Machine

The uBPF virtual machine[12] is a RISC register machine based on:

- Eleven 64-bit registers
  - 9 general purpouse read-write
    - r0 : return values (function calls and exit code)
    - r1-r5: scratch registers used for function call arguments. Upon entering execution of a BPF program, register r1 initially contains the context for the program.
    - r6-r9: not modified through function calls
  - 1 read-only stack pointer.
  - 1 implicit program counter. It cannot be manipulated explicitely by eBPF instructions. It is managed by the virtual machine (or the JIT compiler) based on the flow of execution.

---

11 See the unofficial eBPF spec for details on the structure of each group of instructions and the Cillium documentation for reference.
12 For an introduction, see the series of blog posts by A.Ratiu at Collabora.

- A fixed-size stack. The stack is basically a portion of memory allocated by the virtual machine. This is the only memory that is directly accessible from eBPF programs.

The size of the stack is a parameter of the virtual machine, together with the maximum number of instructions in a program. They are set by default to 512B and 64K respectively.

To each virtual machine is associated a number of external functions. These are functions, whose code is linked with the process that runs the virtual machine. External functions can be registered with the virtual machine. The virtual machine maintains a function pointer table for the external functions. They can then be called from eBPF programs, by their index in the function pointer table (see call instruction above, and example below).

The virtual machine provides a simple interface for:

- Creating/destroying a VM: allocates or deallocates the space needed to hold the VM state (a pointer to the bytecode instructions, the registered functions table, and flags indicating whether checks are enforced and whether the JIT compiler is used).
- Registering/looking up external functions. A lookup based on an external function name returns its index in the VM registered function table.
- Loading/unloading bytecode in a virtual machine. This consists of defining arrays for the registers and the stack, as well as allocating a buffer for the bytecode (based on the length of the eBPF program). The bytecode of the eBPF program is copied there and the number of instructions associated to the program is intialized (based on the program size). A single eBPF program can be loaded in a virtual machine. Unloading a program consists in deallocating the bytecode buffer and resetting the number of instructions associated to the current program to 0.
- Executing bytecode. After allocating the stack and registers, the virtual machine enters the loop that fetches the next instruction (thanks to the fixed sized instructions, an eBPF program is manipulated as an array of instructions), tests the opcode and executes the associated C instructions (usually one or two C instructions per eBPF opcode).
- Checking that predefined constraints in terms of number of instructions or stack size are respected.

The uBPF virtual machine is called from a loader program, that extracts the eBPF program from a given ELF[13] file. The loader relies on the ELF segment header table to identify the different sections of the ELF file, checks its validity, extracts the text sections and replaces references to the name of external functions by their index in the VM external function pointer table and finally loads the concatenated text sections as bytecode in the VM. On a computational storage device, we should aim to reuse the standard uBPF VM and design a loader that is well suited for our purpose.

13 ELF stands for Executable and Linkable Format. This is the format of object code, executables and shared libaries.

The load/unload/execute interface could be the basis for a transport protocol that exposes the VM interface to a host. This is the topic of the Eid-Hermes project[14]. This could also be the basis for a NVMe command set for computational storage.

4.3.3.4 Compiler support for eBPF

Both clang and gcc can compile C programs into eBPF bytecode (as well as eBPF assembly).

Here is a very simple example of a C file (square.c) containing a single function referencing an external function (regfunc)

```
int square(int num) {
    regfunc(num);
    return num * num;
}
```

And the eBPF assembly obtained from clang (clang -S -target bpf square.c)

```
	.text
	.file	"square.c"
	.globl	square                  # -- Begin function square
	.p2align	3
	.type	square,@function
square:                                 # @square
# %bb.0:
	r2 = r1
	*(u32 *)(r10 - 4) = r1
	r1 = *(u32 *)(r10 - 4)
	*(u64 *)(r10 - 16) = r2
	call regfunc
	r1 = *(u32 *)(r10 - 4)
	r1 *= r1
	*(u64 *)(r10 - 24) = r0
	r0 = r1
	exit
.Lfunc_end0:
	.size	square, .Lfunc_end0-square
                                        # -- End function
	.addrsig
	.addrsig_sym regfunc
```

Any function that is referenced from the eBPF program is considered an external function. For example, any call to function from the standard C library is represented as a call to an external function, that should be registered with the virtual machine executing the eBPF program.

There is no support for passing arguments per value, for variadic functions or polymorphic types. As a result, there is no support for compiling C++ programs into eBPF.

4.3.3.5 Data movement

A characteristics of computational storage programs is that they are used to process stored data. Their input is the result of data movement from storage media or storage drives to the computational storage device. Their output should be moved from the computational storage device to the host or possibly to storage drives. Such data movements require (a) the allocation

14 https://github.com/Eideticom/eid-hermes

of buffer space for storing the moved data (or the data to be moved) and (b) access to peripherals and DMA engines to perform the transfers. As we have seen in the previous section, these functions, usually managed by the C standard library or external libraires, are not part of eBPF programs.

We identify the following cases for data movement:

1. Data movement between computational storage and associated storage drives.
   a. Data movement is encapsulated within registered functions, which are responsible for allocating memory and handling I/Os.
   b. Data movement is handled through P2P DMAs[15] set up from the host.
2. Data movement between computational storage and host.
   a. The computational storage device exposes an NVMe I/O command set that enables the host to read and write data to computational storage. The namespace is organized and made available to eBPF program through registered functions.

4.3.3.6 Design Space for Computational Storage

Figure 8 below illustrates a scenario involving eBPF bytecode upload. eBPF code is generated from C functions on the host and offloaded to the computational storage processor that runs a uBPF VM. Two registered functions F1 and F2, accessible from eBPF programs, embed access to directly attached SSDs. The CSP defines a NVM I/O command set exposing the result of eBPF programs as LBA ranges in the context of a block device interface.

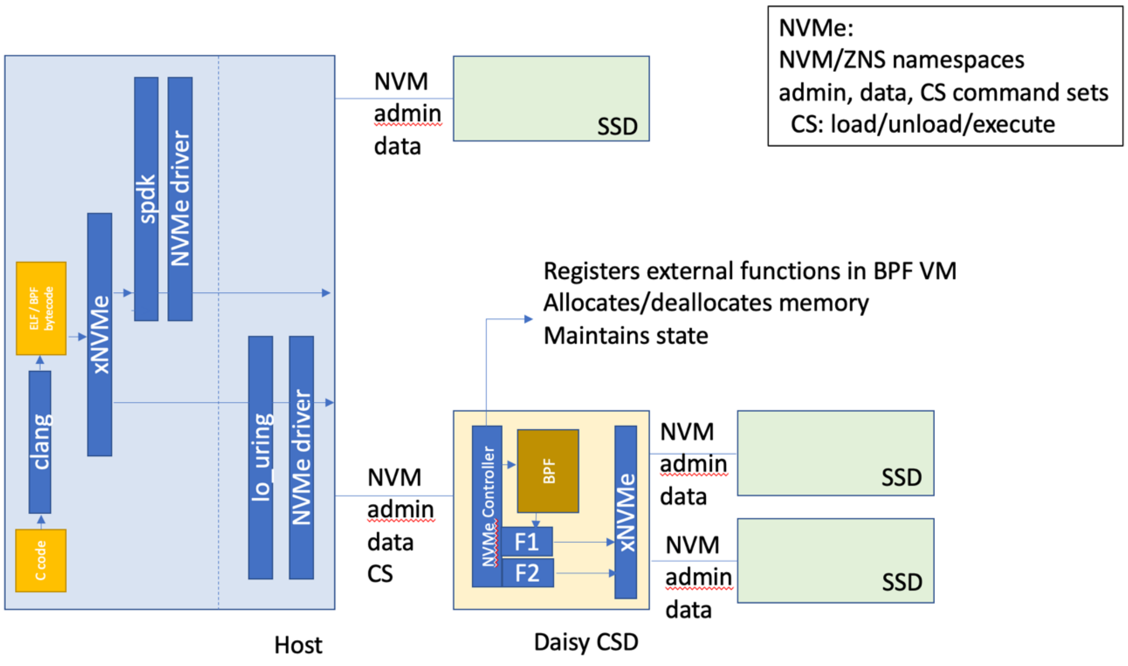


Figure 7: eBPF code offload

15 https://www.kernel.org/doc/html/latest/driver-api/pci/p2pdma.html

As we wrote earlier, eBPF bytecode can define new ways to access pre-installed software (OS, container) and hardware-accelerated (FPGA bitstream) components – made available as registered functions. We make two additional remarks:

1. A generic mechanism to load/unload/execute eBPF programs makes it possible to expose any new SSD interface to a host without modifying the NVMe protocol. In this case, the offloaded eBPF program can be limited to the call of an external function.
2. Access to resources associated to the Computational Storage Processor, such as memory, direct-attached SSDs or other peripherals, must be mediated through registered functions. If those resources are shared across several virtual machines, then the registered functions should provide concurrency guarantees.

4.4 Take-Aways

We identified the various abstractions made available by storage devices (block, zones, files, key-value) and we described the diversity of I/O frameworks that can be used to submit I/Os (posix, aio, libaio, io_uring, spdk, ioctls). We presented xNVMe as a uniform abstraction for accessing NVMe devices regardless of the underlying abstraction or I/O framework.

We presented existing work for programming computational storage devices with a special emphasis on code shipping with eBPF.

5 Storage Configuration

This section goes over the characteristics of different storage tiers focusing on the implications on DAPHNE's integrated data analysis pipelines.

5.1 Storage Tier Characteristics

The storage infrastructure in both HPC clusters and the cloud offer a tiered architecture each serving different purposes and being supported by different storage devices and software frameworks. Mechanisms to move data across storage tiers of different characteristics are similar to data moving across processor caches, DRAM, and persistent storage.

While the names of the tiers and at which granularity they are grouped may change, it is common to consider at least three distinct tiers.

1. ***Performance tier.*** As the name hints, this tier is for keeping and processing the data that is needed frequently by the application (i.e., hot data), but stored for better cost-performance. Access to such data is latency critical. Therefore, the storage devices for this tier are typically NVMe SSDs with SLC/MLC NAND or 3D-Xpoint (see Section 3.1.1).
2. ***Capacity tier:*** The capacity tier offers storage space with a lower price/capacity, as well as lower access latency, for keeping data that is not as latency critical to the application. This tier is composed of HDDs, SATA SSDs, and NVMe SSDs with TLC/QLC/PLC (see Sections 3.1.1 and 3.1.2).
3. ***Archival tier:*** The archival tier targets data that is either too big to even fit in the capacity tier within a reasonable price budget or data that has mainly archival value and is very rarely needed (if needed at all). The storage medium of choice for the archival tier is tape (see Section 3.1.3).

The interactions with data stored in any tier can rely on any of the abstractions presented in Section 4.2. Direct access through I/Os is preferable for the performance tier since latency is of the essence. In addition, raw access to storage devices allows further performance optimization through customizing accesses to storage device based on the application workload. On the other hand, distributed storage systems are preferable for the capacity and archival tiers, as they reduce the headache of reliably managing data on the storage resources at these tiers.

When the application relies on the interfaces and abstractions provided by the storage software frameworks (e.g., Lustre, Ceph, S3) rather than directly using raw access to blocks or zones, there is no room for low-level customization based on device and application characteristics, unless one day the parallel file systems or object stores add hooks to have raw access to storage devices (e.g., customizing number of zones and managing their write-pointers on a ZNS device over Ceph).

Each of these tiers can also have computational storage capabilities that are exposed behind an abstraction. For example, AWS AQUA offers filter functionality for applications that use Amazon Redshift as part of the performance tier. In the future, one can envision that cloud providers offer such services across tiers with a set of pre-defined registered functions to a wider variety of systems behind a well-defined abstraction. Some services may even go a step further and allow applications to offload code as discussed in Section 4.3.2.

5.2 DAPHNE Storage Configuration

A given HPC system has native support for a range of storage systems that are arranged in tiers as described above. For instance, the Vega HPC system at U.Maribor provides (i) local NVMe SSDs on compute instances, (ii) Lustre as performance storage tier and (iii) Ceph as capacity storage tier.

Given the available storage space, programmers should proceed to capacity sizing. They should determine the amount of storage space they need (or can afford) from the native storage systems. The capacity quotas to allocate at each tier depends on the budget of the end-user vs the price/capacity costs of the storage infrastructure being used in addition to the expected data size, data access patterns, and the latency requirements of the application.

When applying for resources on a HPC system, a fixed amount of storage space is requested together with compute resources (on CPU, GPU and possibly FPGA). Local storage on compute nodes can be used directly or in the context of a distributed file system added as modules available to some users or they can be installed on compute instance by users through containers (e.g., HDFS).

This available storage space can be represented as a table storing for each system:

- A type (e.g., Lustre, Ceph, GPFS, NVMe SD)
- An abtraction (e.g., file, block, zones)
- A name (e.g., file system path, device name)
- An amount of available space

To support a particular data access abstraction, IO framework, or storage software framework, the storage back-end matching the interface offered by that abstraction and framework needs to be implemented by DAPHNE. However, since extensibility is one of the core design goals, DAPHNE can add support for these frameworks on a need basis.

The choice of where each data item resides can be determined by a data caching and/or placement policy that is part of the DAPHNE storage configuration tool. For example, the data that should be re-accessed within minutes, hours, days can be kept in the performance, capacity, or archival tiers, respectively. For an application where newer data is accessed more frequently, as a data item becomes older, the access to it will become rarer. Then, it can be gradually moved from performance tier to capacity to eventually archival. Regardless of the data placement policy adopted by DAPHNE, the storage configuration tool must keep metadata to keep track of which data resides where and accessed using what type of abstraction. In the case of more dynamic data management policies, this metadata should also include the access characteristics of the data.

Finally, if a computational storage device or service is available, it is associated to SSDs and it is characterized by a code shipping mechanism (OS image, container, bitstream, eBPF). Computational storage devices supporting eBPF code shipping should also expose the registered functions that they can be accessed from eBPF programs. Then, such capabilities can be utilized based on a cost-model and optimization step that is part of the DAPHNE compiler or runtime or both.

6 Conclusion

All storage devices incorporate compute and storage capabilities. Computational storage denotes devices that can be programmed to fit the needs of a given application workload.

This report first surveyed how various workloads access stored data and identified the promise of computational storage with respect to improved cost-performance, scalability, and resource utilization. It then presents the storage landscape from an architecture and programmer viewpoints, with a focus on computational storage.

While the storage landscape is quite complex, each HPC sytem supports a few native solution organized in local storage on compute nodes, performance tier, capacity tier and possibly an archival tier. When accessing resources on a HPC system, a fixed amount of storage space is requested depending on The capacity quotas to allocate at each tier depends on the budget of the end-user vs the price/capacity costs of the storage infrastructure being used in addition to the expected data size, data access patterns, and the latency requirements of the application.

While multiple frameworks have been defined for submitting I/Os to local SSDs or a file system, we presented xNVMe as a uniform interface that is simple and efficient.

We identified four computational storage devices that are specially relevant in the context of DAPHNE and we presented existing work for programming computational storage devices with a special emphasis on code shipping with eBPF.

7 References

[1] J. Do, I. L. Picoli, D. Lomet, and P. Bonnet, 'Better database cost/performance via batched I/O on programmable SSD', *The VLDB Journal*, vol. 30, no. 3, pp. 403–424, May 2021, doi: 10.1007/s00778-020-00648-z.

[2] M. Torabzadehkashi, A. Heydarigorji, S. Rezaei, H. Bobarshad, V. Alves, and N. Bagherzadeh, 'Accelerating HPC Applications Using Computational Storage Devices', in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Zhangjiajie, China, Aug. 2019, pp. 1878–1885. doi: 10.1109/HPCC/SmartCity/DSS.2019.00259.

[3] H. Choe *et al.*, 'Near-Data Processing for Differentiable Machine Learning Models', *arXiv:1610.02273 [cs]*, Apr. 2017, Accessed: Feb. 23, 2021. [Online]. Available: http://arxiv.org/abs/1610.02273

[4] B. Samynathan, 'Computational Storage For Big Data Analytics', 2019. Accessed: Dec. 11, 2020. [Online]. Available: http://www.adms-conf.org/2019-camera-ready/bala_adms19.pdf

[5] W. Cao *et al.*, '{POLARDB} Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database', 2020, pp. 29–41. Accessed: Mar. 04, 2021. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/cao-wei

[6] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, 'Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics', in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb. 2019, pp. 430–437. doi: 10.1109/EMPDP.2019.8671589.

[7] A. Lerner and P. Bonnet, 'Not your Grandpa's SSD: The Era of Co-Designed Storage Devices', in *Proceedings of the 2021 International Conference on Management of Data*, New York, NY, USA, Jun. 2021, pp. 2852–2858. doi: 10.1145/3448016.3457540.

[8] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, 'Active Disks for Large-Scale Data Processing', *Computer*, vol. 34, no. 6, pp. 68–74, Jun. 2001, doi: 10.1109/2.928624.

[9] Gray, Jim, 'Put Everything in the Disk Controller'. NASD Talk, Jun. 08, 1998. [Online]. Available: https://jimgray.azurewebsites.net/talks/Gray_NASD_Talk.ppt

[10] R. Balasubramonian *et al.*, 'Near-Data Processing: Insights from a MICRO-46 Workshop', *IEEE Micro*, vol. 34, no. 4, pp. 36–42, Jul. 2014, doi: 10.1109/MM.2014.55.

[11] 'SNIA Draft Technical Work available for Public Review | SNIA'. https://www.snia.org/tech_activities/publicreview (accessed Feb. 24, 2021).

[12] A. Barbalace and J. Do, 'Computational Storage: Where Are We Today?', presented at the Conference on Innovative Data Systems Research 2020, Jan. 2021. Accessed: Nov. 09, 2021. [Online]. Available: https://www.research.ed.ac.uk/en/publications/computational-storage-where-are-we-today

[13] 'Best Practice Guide - Parallel I/O, February 2019', *PRACE*. https://prace-ri.eu/training-support/best-practice-guides/best-practice-guide-parallel-io/ (accessed Nov. 09, 2021).

[14] Z. Liang, J. Lombardi, M. Chaarawi, and M. Hennecke, 'DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory', in *Supercomputing Frontiers*, Cham, 2020, pp. 40–54. doi: 10.1007/978-3-030-48842-0_3.

[15] 'What is Cloud Storage? | AWS', *Amazon Web Services, Inc.* https://aws.amazon.com/what-is-cloud-storage/ (accessed Nov. 09, 2021).

[16] 'Azure Cloud Storage Solutions and Services | Microsoft Azure'. https://azure.microsoft.com/en-us/product-categories/storage/ (accessed Nov. 09, 2021).

[17] 'Cloud Storage', *Google Cloud*. https://cloud.google.com/storage (accessed Nov. 09, 2021).

[18]'Alibaba Cloud Storage: Intelligent Storage Service', *AlibabaCloud*. https://www.alibabacloud.com/product/storage (accessed Nov. 09, 2021).

[19]T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, 'All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications', 2014, pp. 433–448. Accessed: Nov. 09, 2021. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai

[20]J. Axboe, 'Efficient IO with io_uring', Linux Kernel Report, 2019. Accessed: Dec. 11, 2020. [Online]. Available: https://kernel.dk/io_uring.pdf

[21]Lund, Simon, P. Bonnet, and J. Gonzalez, 'xNVMe: The Narrow Waist of the modern I/O Stack'.

[22]I. F. Adams, J. Keys, and M. P. Mesnier, 'Respecting the block interface – computational storage using virtual objects', presented at the 11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19), 2019. Accessed: Nov. 09, 2021. [Online]. Available: https://www.usenix.org/conference/hotstorage19/presentation/adams

[23]M. Bjørling, J. Gonzalez, and P. Bonnet, 'LightNVM: The Linux Open-Channel SSD Subsystem', in *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, 2017, pp. 359–374. Accessed: May 26, 2020. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[24]I. L. Picoli, N. Hedam, P. Bonnet, and P. Tözün, 'Open-Channel SSD (What is it Good For)', 2020. Accessed: May 26, 2020. [Online]. Available: http://cidrdb.org/cidr2020/papers/p17-picoli-cidr20.pdf

[25]M. Bjørling, J. Axboe, D. W. Nellans, and P. Bonnet, 'Linux block IO: introducing multi-queue SSD access on multi-core systems', in *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013*, 2013, p. 22:1-22:10. doi: 10.1145/2485732.2485740.

[26]N. Borić, H. Gildhoff, M. Karavelas, I. Pandis, and I. Tsalouchidou, 'Unified Spatial Analytics from Heterogeneous Sources with Amazon Redshift', in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, Jun. 2020, pp. 2781–2784. doi: 10.1145/3318464.3384704.

[27]L. A. N. L. Energy Operated by Los Alamos National Security, LLC, for the U. S. Department of, 'Los Alamos announces details of new computational storage deployment'. https://www.lanl.gov/discover/news-release-archive/2020/November/1116-computational-storage.php (accessed Mar. 04, 2021).

[28]Z. István, D. Sidler, and G. Alonso, 'Caribou: intelligent distributed storage', *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1202–1213, Aug. 2017, doi: 10.14778/3137628.3137632.

[29]J. Do *et al.*, 'Cost-effective, Energy-efficient, and Scalable Storage Computing for Large-scale AI Applications', *ACM Trans. Storage*, vol. 16, no. 4, p. 21:1-21:37, Oct. 2020, doi: 10.1145/3415580.

[30]'Willow | Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation'. https://dl.acm.org/doi/10.5555/2685048.2685055 (accessed Feb. 24, 2021).

[31]B. Gu *et al.*, 'Biscuit: A Framework for Near-Data Processing of Big Data Workloads', in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, South Korea, Jun. 2016, pp. 153–165. doi: 10.1109/ISCA.2016.23.

[32]Z. Ruan, T. He, and J. Cong, 'INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive', p. 17.

[33]W. Huang and M. Paradies, ‘An Evaluation of WebAssembly and eBPF as Offloading Mechanisms in the Context of Computational Storage’. [Online]. Available: https://arxiv.org/abs/2111.01947

[34]Y. J. Wu *et al.*, ‘BPF for storage: an exokernel-inspired approach’, *arXiv:2102.12922 [cs]*, Feb. 2021, Accessed: Nov. 26, 2021. [Online]. Available: http://arxiv.org/abs/2102.12922

[35]M. Jung, ‘OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices’, 2020, pp. 649–656. Accessed: Dec. 02, 2020. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/jung

8 Appendix 1: PCIe

PCIe is a layered network protocol, based on requests/responses (denoted transactions), layered atop a packet-based data link protocol and physical connections organized as a collection of lanes. Each lane is a pair of unidirectional, serial, point-to-point connections.

The PCIe fabric is organized as a tree, with a root complex and mutiple endpoints connected directly or via switches. Each PCIe device has an identifier (PCI ID) composed of a vendor ID and a device ID.

Each device is associated to a memory-mapped region of the host address space, defined through the Base Address Registers (BAR). This region of memory includes a collection of registers used for configuration and operation. For instance, at boot time, PCI devices do not have addresses assigned to them. The host bios must thus enumerate all devices on the PCI fabric and initialize the BAR configuration register for each device found.

PCIe supports message-signaled interrupts (MSI-X), so individual interrupts can be addressed to specific host cores.

9 Appendix 2: NVMe

Hosts and devices communicate through pairs of submission/completion queues. The queues are located in the memory-mapped address space either on the host or on the device (queues created in different cores can be used without coordination). The driver puts commands on the submission queue and writes a doorbell register when commands are ready to be executed. The controller accesses the commands from the queue. Data is transferred between a host's memory and an SSD via Direct Memory Transfer (DMA). After commands are executed, the controller puts the status for the completed command on the associated completion queue. Multiple submission queues can be associated to the same completion queue. The controller then notifies the host that completion notifications are available through a doorbell register. This streamlined I/O path reduces software overhead and is faster than previous storage interconnects. For more details, we refer readers to the excellent overview of NVMe provided by Myoungsoo Jung in his paper on Open-Express [35].

An NVMe controller may support administration commands, multiple namespaces, and zero or more I/O Command Sets defining the operations that can be performed with the namespace. Administrative commands include the creation and deletion of submission/completion queues, as well as primitives for device identification, or getting log-pages, device capabilities, and features. A namespace is the encapsulation of a resource that is made available to hosts via a I/O

Command Set. Directives enable host and device to exchange command meta-data, e.g., streams.

The NVM I/O Command Set is associated to a logical representation of non-volatile memory as a collection of blocks. It defines familiar commands such as read and write, with additions such as Write-Zeroes, providing the means to set LBAs to zero without transferring a payload of all zeroes from host to device. The Zoned Command Set, relies on a logical representation where logical blocks are partitioned into zones. It establishes constraints that logical blocks must be written sequentially within a zone and that zones must be reset before they are written. It defines the zone state that must be communicated from device to host. It also defines the append command. The Key Value Command Set abstracts the underlying resource as a pool of bytes which can be allocated in key-value pairs. Access to key-value pairs is done through store and retrieve commands, with additional commands to check for the existence, delete, and list keys in the namespace. The standardisation of computational storage in NVMe is under way. We envision that (a) the underlying resource, for which a namespace provides encapsulation, is a computational unit rather than non-volatile memory, (b) the abstraction provided by the Command Set will be close to that of a JIT compiler over a virtual instruction set architecture.

NVMe is now defined over PCIe, Ethernet and Infiniband fabrics. It has been recently suggested to extend NVMe to hard disk drives [1, 18] to unify all storage under NVMe in data centers. In summary, NVMe makes it possible to execute traditional read/writes faster, it makes it possible for the host software to shape streams of reads and writes, and it makes it possible for host software to issue new commands on new storage abstractions. To leverage NVMe SSDs, system designers need an efficient and flexible I/O interface.

Lerner et al. propose the following parameters to characterize I/O workloads on NVMe drives [7]:

- I/O pattern (sequential/random, read/write/mixed)
- I/O size
- Number of cores submitting I/Os
- Number of in-flight I/Os per core
- Other parameters (circularity, latency sensitiveness, burstiness)

10 Appendix 3: NAND Flash

NAND flash relies on arrays of floating-gate transistors, so-called cells, to store bits. Shrinking transistor size has enabled increased flash capacity. SLC flash stores one bit per cell. MLC and TLC flash store 2 or 3 bits per cell, respectively, and there are four bits per cell in QLC flash. For 3D NAND, increased capacity is no longer tied to shrinking cell size but to flash arrays layering.

There are three fundamental programming constraints that apply to NAND: (i) a write command must always contain enough data to program one (or several) full flash page(s), (ii) writes must be sequential within a block, and (iii) an erase must be performed before a page within a block can be (re)written. The number of program/erase (PE) cycles is limited. The limit depends on the type of flash: 10^2 for TLC/QLC flash, 10^3 for MLC, or 10^5 for SLC.

Additional constraints must be considered for different types of NAND flash. For example, in multi-level cell memories, the bits stored in the same cell belong to different write pages, referred to as lower/upper pages. The upper page must be written before the lower page can be read successfully. The lower and upper page are often not sequential, and any pages in between must be written to prevent write neighbor disturbance.

As a result, we consider two key parameters for characterizing SSD capabilities: NAND type and unit of read/write/erase.