UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
SoC-Based In-Storage Processing: Bringing Flexibility and Efficiency to Near-Data Processing

Permalink
https://escholarship.org/uc/item/40m4t7gp

Author
Torabzadehkashi, Mahdi

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

SoC-Based In-Storage Processing: Bringing Flexibility and Efficiency to Near-Data Processing

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Mahdi Torabzadehkashi

Dissertation Committee:
Professor Nader Bagherzadeh, Chair
Professor Phillip Sheu
Professor Jean-Luc Gaudiot

2019

© 2019 Mahdi Torabzadehkashi

DEDICATION

This one to
The most important things in the world
family and love

"He who knows not and knows not he knows not; he is a fool - shun him; He who knows not and knows he knows not; he is simple - teach him; He who knows and knows not he knows, he is asleep - wake him; He who knows and knows he knows; he is wise - follow him"
poem, by Ibn-i Yamin (1286-1368).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The Ph.D. study was not a short-term sprint; instead, it was a tough marathon wherein a lot of people helped in the path. When I started my Ph.D. studies, I was excited and expected a smooth run from start to end. However, later, I found out this path was full of bridges over troubled water, and remembering them makes me thankful to all the people who helped me cross those bridges.

I would like to acknowledge my advisor Professor Bagherzadeh, who led me throughout my Ph.D. studies. I always knew he was in his office and would help me whenever I needed it. I would also like to deeply thank Dr. Vladimir Alves and all the team members in NGD Systems, Inc. I could never explain the awesome experience that I had with this company during my internship. Dr. Alves is one of the most knowledgeable individuals I have ever met, and without his support, this dissertation would not even exist. I also want to thank my friends, Siavash Rezaei, Ali Heydarigorji, and Dr. Hosein Bobarshad, for their valuable comments and contributions while preparing the papers we published together.

CURRICULUM VITAE

Mahdi Torabzadehkashi

EDUCATION

Doctor of Philosophy in Computer Engineering **2019**
University of California, Irvine (UCI) *Irvine, California*

Master of Science in Computer Architecture Engineering **2014**
Sharif University of Technology (SUT) *Tehran, Iran*

Bachelor of Telecommunication Engineering **2010**
Shahr-e-Rey Azad University *Tehran, Iran*

RESEARCH AND WORK EXPERIENCE

Internship in the computational storage R&D team **2016–2019**
NGD Systems, Inc. *Irvine, California*

Graduate Research Assistant **2014–2015**
University of California, Irvine *Irvine, California*

Graduate Research Assistant **2011–2014**
VLSI Laboratory, Sharif University of Technology *Tehran, Iran*

Head of the Robotics Team **2007–2010**
Shahr-e-Rey Azad University *Tehran, Iran*

TEACHING EXPERIENCE

Teaching Assistant **2012–2013**
Data Communications,
Digital Electronic Lab,
Digital System Design Lab
Sharif University of Technology *Tehran, Iran*

Teaching Assistant **2009–2010**
Digital System Design,
Computer Architecture
Shahr-e-Rey Azad University *Tehran, Iran*

REFEREED CONFERENCE PUBLICATIONS

Accelerating HPC Applications Using Computational Storage Devices — **August 2019**
The 21st IEEE International Conference on High Performance Computing and Communications (HPCC)

Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics — **February 2019**
The 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)

CompStor: An In-storage Computation Platform for Scalable Distributed Processing — **May 2018**
The 32nd IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

ABSTRACT OF THE DISSERTATION

SoC-Based In-Storage Processing: Bringing Flexibility and Efficiency to Near-Data Processing

By

Mahdi Torabzadehkashi

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2019

Professor Nader Bagherzadeh, Chair

Data are among the most valuable assets in the modern world, and they have caused a revolutionary stage in human life. Nowadays, companies make knowledge-based decisions by analyzing a huge volume of data, super-scale data centers are used to process customers' data to suggest products to them, government services rely on the data people provide to them, and there are many similar cases wherein data are used as an important asset. Data are originally stored in storage systems. To process data, application servers need to fetch the data from storage units, which imposes the cost of moving the data to the system. This cost has a direct relationship to the distance of the processing engines from the data, and this is the key motivation for the emergence of distributed processing platforms such as Hadoop, which bring the process closer to the data.

In-storage processing (ISP) pushes the "bring the process to data" paradigm to its ultimate boundaries by utilizing processing engines inside the storage units to process data. The architecture of modern solid-state drives (SSDs) provides a suitable environment for implementing such technology. Thus, this dissertation focuses on SSD architectures that are able to run user applications in-place, which are called computational storage devices (CSDs). In this dissertation, we propose CSD architectures and investigate the benefits of deploy-

ing CSDs for running different applications. This research uses a practical approach that includes building fully functional prototypes of the proposed CSD architectures, developing storage systems equipped with the CSDs, and running different benchmarks to investigate the benefits of deploying the CSDs in the systems. This research proposes two different CSD architectures, namely CompStor and Catalina.

These are the first CSDs to be equipped with a dedicated ISP engine for running user applications in-place that includes a quad-core ARM Cortex-A53 processor together with FPGA- and application-specific integrated circuit (ASIC) based accelerators. The proposed architectures run a full-fledged operating system inside, which provides a flexible environment for running a wide range of user applications in-place. The system-on-chip (SOC) based architecture of Catalina CSD, together with a software stack developed for seamless deployment of the CSD, makes it a platform for the implementation of different ISP concepts and ideas.

To the best of our knowledge, Catalina is the only ISP platform that can be seamlessly deployed in clusters to run distributed applications such as Hadoop MapReduce and message passing interface (MPI) based applications in-place without any modifications to the underlying distributed processing framework. We performed extensive experimental tests using several datasets on both CompStor and Catalina CSDs. The experimental results show up to 2.2x and 4.3x improvements in performance and energy consumption, respectively, for running Hadoop MapReduce benchmarks using Catalina CSDs and up to 5.4x and 8.9x improvements for running 1-, 2-, and 3-dimensional DFT algorithms due to the Neon SIMD engines inside Catalina. Additionally, using FPGA-based accelerators, Catalina CSDs can improve the performance and energy consumption of a highly demanding image similarity search application up to 11x and 7x, respectively.

Keywords— *computational storage*, *in-storage processing*, *near-data processing*, *Catalina*, *CompStor*, *SSD*, *system-on-chip*, *big data*, *Hadoop*, *Faiss*, *HPC*, *DFT*, *image similarity search*

Chapter 1

Introduction

The modern human's life has been technologized, and nowadays, people rely on super-scale applications to receive services such as healthcare, entertainment, government services, and transportation in their day-to-day lives. As the usage of these services becomes universal, people generate more unprocessed data, which increases the demand for more sophisticated data centers and applications. The valuation of the generated data has been highlighted by several research works [1, 2]. According to the well-known 4V's characteristics of big data, the systems need to deal with very large *volumes* of data which are in *various* types, and their *velocity* is more than that of conventional data, while the data's *veracity* is not confirmed [3].

To process data with the mentioned characteristics, the data should frequently move between the storage systems and memory units of the application servers. This high-cost data movement imposes extra energy consumption and potentially degrades the performance of the applications. Therefore, data processing has moved toward a new paradigm: "bring the process to data" rather than moving high volumes of data. Fig. 1.1 compares the traditional "moving data to the process" policy to the "bring the process to data" paradigm.

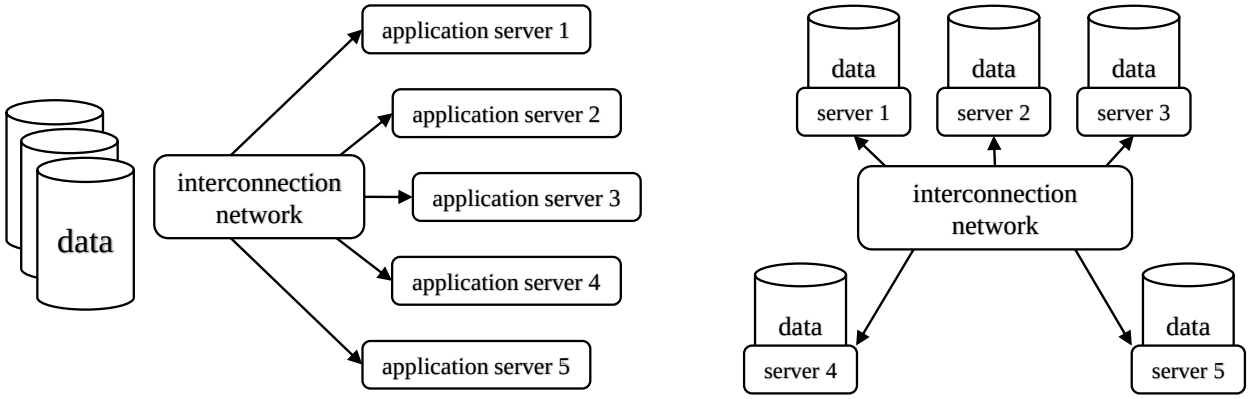


Figure 1.1: Comparing "data move to process" and "bring process near data" paradigms

In modern clusters, nodes are connected via a low-latency and power-hungry interconnect network such as InfiniBand [4] or Myrinet [5]. In such systems, moving data can be more expensive than processing it [6, 7]. In fact, accessing and transferring data from the storage systems to the application servers can be a huge barrier toward reaching compelling performance and energy efficiency. To deal with this issue, some frameworks such as Apache Hadoop [8] provide mechanisms to process data near where they reside. In other words, these frameworks push the process closer to the data to avoid massive data movements between storage systems and application servers.

In-storage processing (ISP) is the concept of pushing the process closer to the data in its ultimate boundaries. This technology proposes utilizing embedded processing engines inside the storage devices to enable them to run user applications in-place, so data do not need to leave the devices to be processed. This technology has been around for a few years. However, the modern solid-state drive's (SSDs) architecture, as well as the availability of relatively powerful embedded processors, makes it more appealing to run user applications in-place. SSDs use flash memory as the storage media and deliver higher data throughput in comparison to hard disk drives (HDDs). Moreover, they contain a considerable amount of processing horsepower in the form of multiple embedded cores for managing the flash memory array and providing a high-speed interface to the host. These processing engines could potentially provide an environment to run user applications. Based on the reasons

mentioned above, this research focuses on modern SSD architecture, and, in the rest of this dissertation, computational storage device (CSD) refers to an SSD that is enabled to run user applications in-place.

In a CSD architecture, an ISP engine is responsible for running user applications. This engine potentially accesses the data stored in the flash memory array through a low-power, high-speed link. Thus, the deployment of such CSDs in systems can increase the overall performance and efficiency.

1.1 Dissertation Objectives and Contributions

Processing user applications inside storage units without sending data to the host processor seems appealing. However, proposing a flexible and efficient CSD architecture comes with the following challenges:

1. **ISP engine:** SSDs come with multiple processing cores to run the conventional SSD controller routines. These cores can be utilized for running user applications as well. However, there are two major problems in utilizing the existing SSD cores for ISP. First, these cores are usually busy doing normal SSD operations, and using them to run user applications can negatively affect the I/O performance of the drive. Second, these processing engines are usually real-time cores such as the ARM Cortex-R series [9], which limits the category of applications that can efficiently run on these cores. In some cases, user applications need major modifications to be able to run on these cores.

2. **Host-CSD communication:** In a CSD architecture, there should be a mechanism for the communication between the host and CSD to submit ISP commands from the host to CSD and receive the results. Regularly, conventional SSDs have one physical

link connected to the host that is designed for transferring data. There are many protocols for sending data through this link such as SATA [10], SAS [11], and NVMe [12]. None of these protocols are designed for sending ISP commands and results. Thus, it is the responsibility of the CSD designer to provide an ISP communication protocol between the host and CSD.

3. **Block-level or filesystem-level data access:** An embedded processing engine inside a CSD has access to the raw data stored on the flash, but the filesystem metadata is in control of the host. As a result, data access inside the storage unit is limited to the block-level data. Therefore, any application running in-place should not expect to be able to access the filesystem-level data. This issue strictly limits the type of programming models available for developing ISP applications as well as the reuse of other applications. Thus, the CSD designer should provide a mechanism to access the filesystem metadata inside the ISP engine so the applications that run in-place can open files, process data, and, finally, create output files to write back the results.

4. **Host-CSD data synchronization:** In a system where a host is equipped with a CSD, potentially, both the host and the ISP engine inside the CSD have access to the same flash memory array simultaneously. In such a system, without a synchronization mechanism, these two machines may not be able to see each other's modifications, which could result in data corruption.

5. **CSDs as an augmentable resource:** Attaching CSDs to a host machine should not limit the host from accessing the data and processing it. The processing horsepower of the CSDs should be an augmentable resource so that the host and CSDs can process data concurrently. If processing an application in the CSD interferes with the host's access to the data, this will dramatically decrease the utilization of the host and the efficiency of the whole system. A well-designed CSD architecture allows the host to access data stored in the flash memory at any time.

6. **Adoptability:** CSDs should provide a flexible environment for running different types of applications in-place. If the ISP engine of a CSD supports very limited programming languages or needs users to rewrite the application based on a specific programming model, this can negatively affect the adoptability of the CSD.

7. **Distributed ISP:** A single CSD with limited processing horsepower may not be able to enhance an application's performance satisfactorily, so in many cases, there should be multiple CSDs orchestrating together to deliver compelling performance improvement. To perform such distributed processing, CSD designers need to provide the required tools for implementing a distributed processing environment among multiple CSDs.

8. **ISP for compute-intensive applications:** Highly demanding applications such as high-performance computing (HPC) algorithms can potentially run inside CSDs. However, to serve this class of applications, CSDs should be able to boost their performance for some specific applications. In other words, the CSD architecture should be customizable enough to run some applications in an accelerated mode. Therefore, CSD designers are required to provide ASIC- or FPGA-based accelerators to run highly demanding applications appropriately.

These challenges are outlined in the literature, and a large number of research works have tried to solve a subset of the challenges mentioned above. However, to the best of our knowledge, there is no ISP solution that addresses all the above challenges. This dissertation aims to investigate these challenges and propose solutions to overcome them. Throughout this research, we will show how each of these challenges can be addressed, and using a practical method, we will explore different architectures and investigate the benefits of the proposed solutions for I/O- and compute-intensive benchmarks in both distributed and non-distributed environments.

The contributions of this research can be summarized as follows:

- We will discuss the challenges of developing efficient CSD architectures and propose solutions to address them.

- We will describe the path that led us to develop an efficient CSD architecture. We propose two CSD architectures in this dissertation, namely CompStor and Catalina. Using a practical approach, we show why the earlier design is not aligned with the ISP core concepts. However, the Catalina CSD meets or exceeds our expectations and shows how CSDs can improve the performance and energy efficiency of the systems.

- Both CompStor and Catalina CSDs were prototyped to show the feasibility of the proposed solutions. Building these CSDs gave us an accurate understanding of the challenges of designing and manufacturing efficient CSD architectures.

- We explored the utilization of FPGA- and ASIC-based accelerators inside CSD architectures for improving the performance and energy efficiency of highly demanding applications, such as image similarity searches on a very large dataset as well as 1D, 2D, and 3D DFT calculations.

- Different platforms equipped with multiple CSDs were built to explore the benefits of the deployment of CSDs in systems. We used several I/O- and compute-intensive applications as well as distributed benchmarks such as Hadoop MapReduce and MPI-based applications to run extensive experimental tests.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 summarizes some notable related works in the literature. In Chapter 3, we first provide an overview of the modern SSD architecture and how ISP technology improves the performance and efficiency of the systems. Then, we propose two CSD architectures, which are called CompStor and

Catalina. Additionally, we discuss the features of the Catalina CSD that make it a platform for implementing different ISP ideas.

Chapter 4 explores the deployment of Catalina CSDs in the Hadoop and MPI-based clusters and reports the results of running the Hadoop MapReduce and HPC benchmarks on the developed platforms that are equipped with up to 16 Catalina CSDs. In this chapter, we show how utilizing Neon SIMD engines for running HPC applications improves performance and energy efficiency considerably. Chapter 5 investigates the utilization of the FPGA-based accelerators for running a highly demanding image similarity search application on the ISP-enabled systems. Finally, Chapter 6 concludes this dissertation and discusses future topics that could extend the contributions of this research.

Chapter 2

Literature Review

Primarily, there are two categories of near-data processing, which are *near main memory processing* and *in-storage processing* [13]. These technologies aim to bring the process closer to data that are stored in different levels of the memory hierarchy. In this research, we focus on the processing data inside storage units, i.e., data do not need to be transferred to the host's memory to be processed.

ISP technology was initially introduced for hard disk drives. Archarya et al. proposed an ISP-enable HDD architecture with a dedicated ISP engine that includes an x86-based processor running at 200 MHz and 16 MB of memory for running user applications in-place [14]. They set the cost constraint for implementing the ISP engine to $100, which was a considerable cost overhead at the time of publishing the paper.

In those early stages, the filesystem challenge, which was discussed in the first chapter of this dissertation, was remarked by Lim et al. [15]. They introduced a filesystem designed for the ISP-enabled HDDs called the active-disk-based data server (ADFS). The ADFS is a filesystem that is implemented partially in the ISP-enabled disk and the host.

There are other papers that explored the ISP-enabled HDDs in the late 1990s [16, 17, 18]. However, these ISP-enabled HDDs could not reach a satisfactory level of improvement and feasibility due to the relatively high manufacturing cost of implementing an ISP engine in the HDDs, the inherent random latency of HDDs, and the limited availability of the processor that could be embedded in the HDDs at that time.

After the emergence of SSDs, ISP technology found a new platform on which to flourish. The SSDs deliver better throughput in comparison with HDDs, especially for random read and write operations. Oftentimes, they have multiple processors inside for conventional flash management and to host interface routines. Such a device can be better extended to run user applications. However, there is a considerable gap between the available data bandwidth inside SSDs and the bandwidth that is provided to the host. This gap is an influential motivation for processing user data in-place. In this dissertation, the SSDs that can run user applications in-place are called CSDs. There are two main categories of the related works: 1.) research works that advocate for the utilization of the general-purpose processors for running user applications inside the CSDs (processor-based CSDs), and 2.) publications that propose using pure FPGA-based accelerators inside the CSDs (FPGA-based CSDs). In addition to these two main groups, there is a third group with a limited number of research works that propose other mechanisms. In this chapter, we will review the notable related works in the two main categories as well as the limited works that use other mechanisms to run user applications inside the storage units.

2.1 Processor-Based CSDs

The majority of processor-based CSD architectures share the same processing resources both for running user applications and for conventional flash management routines. This class of CSDs mainly uses real-time processors that are already available in the SSD architectures [19,

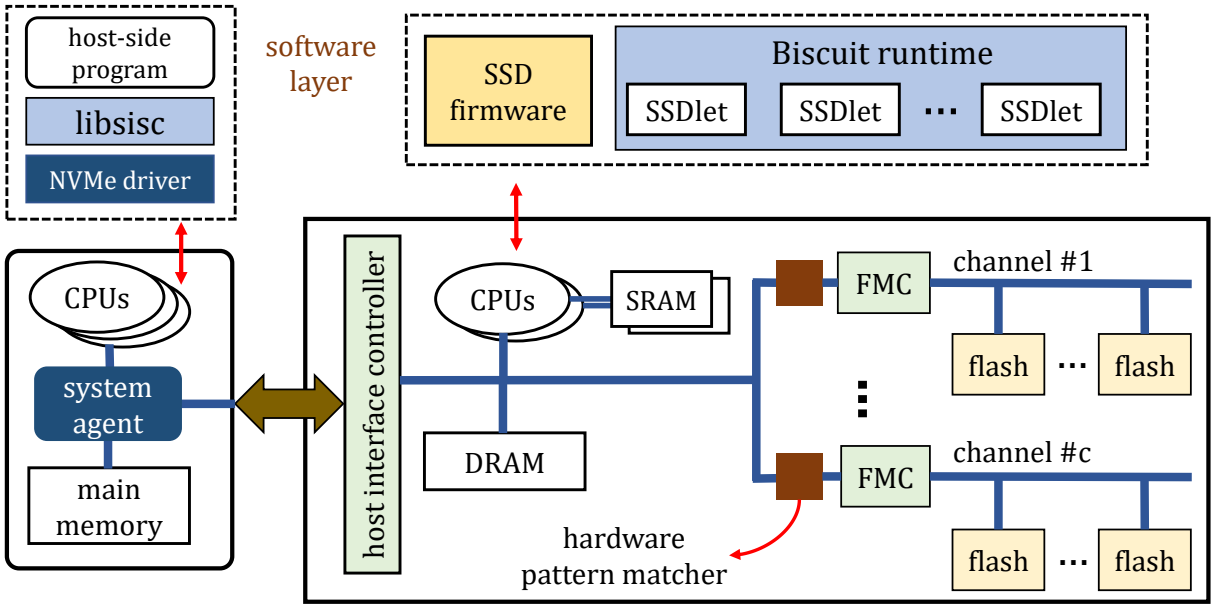


Figure 2.1: Overall architecture of Biscuit CSD

20]. Although utilizing the available processors for ISP purposes increases the utilization of the existing processors in the storage unit, it does not provide enough processing horsepower for compute-intensive tasks, as a user application can interfere with the conventional SSD's I/O operation, and vice-versa. Even worse, many of these architectures are based on 32-bit embedded processors that are not suited for executing complex applications. In other words, using the same real-time processor for two concurrent tasks, user applications and SSD firmware control, may cause a loss in performance in both tasks.

Biscuit uses ARM Cortex-R7 embedded real-time processors together with a set of hardware pattern matcher modules in the storage device to provide an environment to run I/O-intensive applications in-place [19]. Fig. 2.1 (adopted from [19]) shows the overall system of Biscuit, including its hardware and software architecture. The proposed architecture makes a seamless distributed environment for the host and CSD to run the user applications.

In Biscuit, users have to develop the ISP application based on a flow-based programming model [21]. Using a set of APIs (*libsisc* library) that are available in Biscuit, the user needs to develop a set of *SSDlets* that communicate with each other to perform the ISP task. In such a

programming model, the CSDs are slaves to the host, which is responsible for controlling the ISP flow. The software layer of Biscuit supports "dynamic module loading" and "dynamic memory allocation," which are two useful features for running ISP applications.

Each *SSDlet* is a simple C++ program developed using Biscuit's provided APIs. Since the applications have to be developed based on a flow-based programming model, users cannot reuse other types of applications. Biscuit CSD has been prototyped; however, there are major hardware limitations in the prototype such as no cache coherency, a limited amount of fast memory, and no memory management unit (MMU).

Kim et al. proposed a CSD architecture for database scan and join operations [22]. To exploit the full parallelism inside the SSD architecture, they designed per-flash channel processing elements that collaborate with the embedded ARM Cortex-A9 processor to implement the database scan and join operations. In fact, a part of the operations is executed on the data path, while the data is prepared for transmission inside the storage. This design is intended only for two specific operations and cannot be generalized. To evaluate the benefits of the proposed architecture, they used a simulation and modeling approach.

Kang et al. proposed Smart SSD, which supports a modified version of the Hadoop MapReduce programming model to run ISP tasks in CSDs [20]. They used an off-the-shelf Samsung SATA SSD as the hardware platform and did not make any modification to the hardware. However, they modified the SSD controller firmware to support their modified version of the Hadoop framework. Smart SSD does not support any message passing mechanism between the CSDs. Since they only support an extended version of the Hadoop MapReduce programming model, they need to break down an ISP task to *tasklets* and cross-compile them to be able to run them on the Samsung SSD's internal ARM cores. The internal ARM cores are shared between the conventional flash management routines and the ISP engine, so only trusted people can develop ISP applications for Smart SSD.

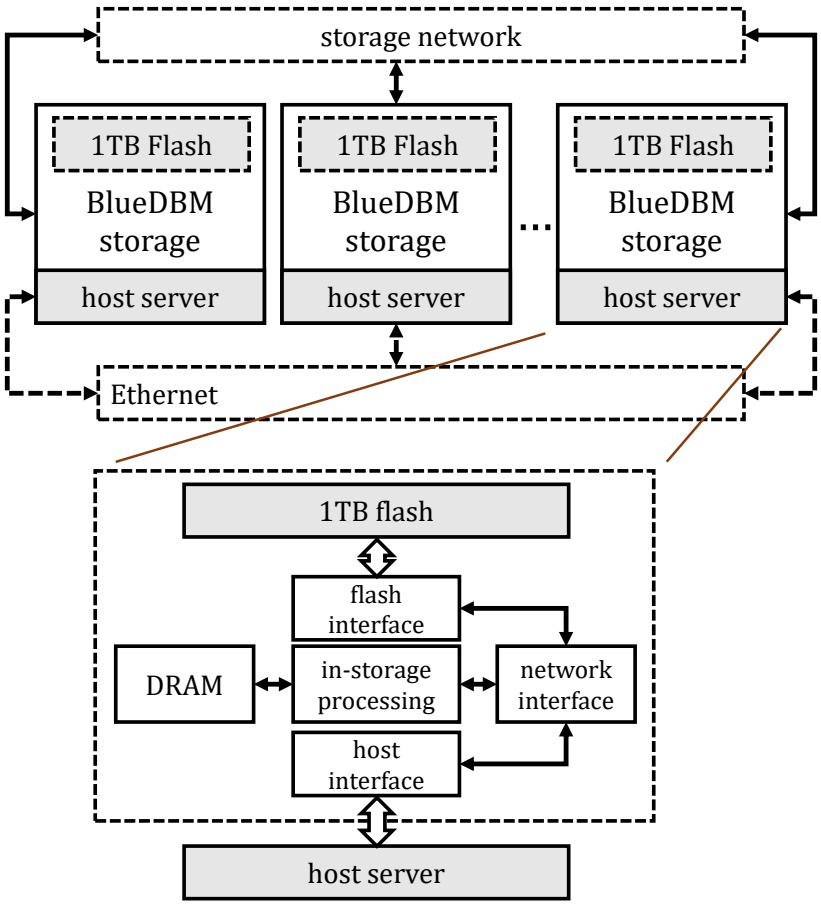


Figure 2.2: BlueDBM overall and node architecture

2.2 FPGA-Based CSDs

The second category covers FPGA-based CSD architectures. While FPGAs potentially give higher horsepower, a pure FPGA design suffers from a lack of generality and flexibility compared to general-purpose processors. Jun et al. proposed an architecture for scalable multi-access flash storage for big data analytics called BlueDBM, wherein the whole flash controller, host interface, and computation unit are implemented on FPGAs [23, 24]. Fig. 2.2 (adopted from [23]) shows the overall architecture of a cluster that is equipped with BlueDBM CSDs, as well as the architecture of a node.

Each server in Fig. 2.2 is attached to a BlueDBM CSD. Inside each CSD, the ISP unit has access to four other modules: the flash interface, network interface, host interface, and

DRAM module. The cluster has two network interconnects, which are the regular fabric between the hosts, and an inter-controller network (storage network). The storage network allows one CSD controller to access the other CSDs' data. In other words, the flash interface can be accessed by the local ISP engine (ISP unit in Fig. 2.2), the local host, and another CSD controller concurrently. From a performance point of view, this structure provides a scalable ISP-enabled cluster architecture since the storage network provides a high-speed link to other CSDs' data. However, due to the existence of two networks (the regular fabric between the hosts and the storage network), when there are a large number of ISP-enabled nodes, the physical scalability of the proposed architecture is questionable.

The BlueDBM filesystem is limited to the refactored file system (RFS) [25]. Normally, the SSD controller handles the internal flash translation layer (FTL), and the internal characteristic of the flash memory is invisible to the host. Unlike conventional filesystems, RFS handles some functionalities of the FTL, such as translating the virtual addresses to the physical flash block numbers. BlueDBM needs the host to use an RFS-based filesystem, and this limitation could negatively affect the adoptability of the proposed solution.

Vincon et al. proposed nativeNDP [26] as an FPGA-based CSD architecture that is able to run an *R script* [27] inside the storage units of the nodes in a Ceph cluster [28]. In fact, they designed a custom *R* plugin to interact with the Ceph cluster. The nativeNDP uses a NoFTL-KV storage engine [29] in the host, which provides the physical address of the flash memory blocks inside the host. To evaluate the performance of nativeNDP, they used a simulation environment and ran benchmarks using synthetic datasets.

Another example of an FPGA-based CSD architecture is RISP [30]. The RISP CSD includes a reconfigurable unit (RU), which is composed of a processing cell and an NVM controller on each flash channel, as well as a public processing cell shared by all the channels. In fact, in the proposed architecture, a set of memory chips on a flash channel together with the corresponding NVM controller and processing cell form a RISP channel, which is the

minimum processing element of the RISP architecture. The public processing cell can access all the RISP channels. All the components mentioned above are implemented in FPGA.

Overall, the modification of an FPGA-based CSD architecture to provide new functionalities is time-consuming and error-prone and includes RTL design, synthesizing, and bitstream generating. Additionally, since the functionality of a pure FPGA-based CSD is limited, in some scenarios, pre-processing on the host system is required before invoking FPGA-based accelerators in the CSD [31]. This can impose unnecessary data transfer to the system that has a significant interference with the core concept of ISP technology.

2.3 Other CSD Architectures

Although the aforementioned categories cover most of the related works, there are a limited number of papers that do not belong to either of those two main categories. Cho et al. proposed a CSD architecture called XSD that uses a graphics processing unit (GPU) as the ISP engine [32]. They provided an API set to the user for uploading the task to the CSD based on a modified version of the MapReduce programming model. They used a simulation-based approach to evaluate the performance of the proposed GPU-based CSD and reported that the GPU-based CSD is considerably faster than processor-based CSDs. However, only applications that are developed based on the modified version of MapReduce are off-loadable to the proposed CSD.

PRINS [33] is an ISP-enabled storage based on resistive content-addressable memories (ReCAM) for machine learning applications. This CSD architecture does not follow the *von Neumann* model. There is one associative processing unit per ReCAM memory row and a microcontroller that controls these processing units; these elements form the ISP engine. The code of the PRINS microcontroller is developed manually in the assembly language,

and there is no data coherency between the host's CPU and the ISP engine. Thus, for the sake of data coherency, PRINS does not allow the host's CPU to access the storage during in-storage task executions. This is a serious limitation that can decrease the host's CPU utilization significantly.

2.4 Summary

After reviewing the related works, we can now summarize the main challenges that have not been addressed in the previous works. In almost all the works, the major challenge is adoptability. In other words, in the proposed solutions, data is only available at the block-level to the ISP engine of the CSDs; therefore, applications cannot deal with the filesystem concepts, as needed for conventional programming. In some of the related works [23, 24, 26], the CSD architects tried to solve this problem by dividing the flash memory management tasks between the host's operating system (OS) and the CSD controller. Using this approach, the internal physical block addresses of the flash memory are exposed to the host's operating system, and the user can send the block addresses alongside the main ISP task to the CSD. As a result, the ISP engine can access the physical blocks to run the ISP application on the data. However, to the best of our knowledge, all the available CSDs require major modifications to the conventional applications to be adapted to the CSD architectures. In some CSDs, the applications have to be developed from scratch based on a specific programming model [19].

In this research, we address these issues by porting a full-fledged Linux operating system into the ISP engine and developing the necessary tools to execute a wide range of applications in-place without modifying them. Another issue is the imposed limitation on the host's data access during execution of the ISP applications [33]. This problem is also addressed in this research.

Table 2.1: Comparison between notable related works and one of the architectures proposed in this research

	ISP engine	**programmability**	**filesystem support in ISP engine**	**ISP-supported distributed processing**
Biscuit [19]	2x ARM Cortex-R7 (shared)	medium (Biscuit API set)	not supported	not supported
Scan & Join [22]	ARM Cortex-A9 (shared)	limited (database scan and join)	not supported	not supported
Smart SSD [20]	2x ARM cores (shared)	medium (MapReduce)	not supported (object-based)	A modified version of Hadoop MapReduce
BlueDBM [23]	FPGA (dedicated)	limited (RTL design required)	not supported (RFS-based)	inter-controller network
NativeNDP [26]	FPGA (dedicated)	medium (R language)	not supported (NoFTL-KV)	limited to Ceph
RISP [30]	FPGA (dedicated)	limited (RTL design required)	not supported	not supported
XSD [32]	GPU	medium (MapReduce)	not supported	A modified version of MapReduce
PRINS [33]	associative processing units	limited (non-von Neumann)	not supported	not supported
Catalina	**4x ARM A53 + FPGA (dedicated)**	**full (e.g. C++, Python)**	**full support**	**full support (e.g. MPI / Hadoop)**

Table 2.1 compares one of the proposed CSD architectures in this dissertation, Catalina, to the CSD architectures proposed by the notable related works. In this table, the first column shows the names of the CSD architectures. The second column indicates the type of ISP engine for each CSD. The third column shows the adaptability and programmability of the architectures. To the best of our knowledge, among all the previous works, Catalina is the first CSD that includes an ISP engine that can execute, potentially, any application with no modification. The fourth column represents the capability of accessing data at the filesystem-level inside the ISP engines. Finally, the fifth column shows the potentials of the proposed CSD architectures in distributed environments.

Chapter 3

A Practical Approach to Proposing CSD Architectures

We mentioned the challenges of proposing a well-designed CSD architecture in the first chapter. Based on these challenges, we set eight design goals to propose an efficient and flexible computational storage platform. The design goals are as follows:

1- A desired CSD architecture should avoid using real-time processors that were originally intended to run conventional flash management routines for running user applications in place; instead, it should contain an ISP-dedicated application processor to provide a flexible environment to run user-applications without negatively affecting normal I/O operations. **2**- There should be a TCP/IP link between the host and the ISP engine inside the CSD so that the applications running on the host and CSD can communicate with each other. **3**- The ISP engine should have access to the filesystem-level data so that it can read files that are stored in the flash memory, process them, and generate output files. **4**- Since both the host and the ISP engine have access to the same flash memory at the filesystem-level, there should be a synchronization mechanism such as clustered filesystems to ensure the integrity

of the data. **5**- Both the host and the ISP engine should be able to interact with the flash storage concurrently. This makes the CSD an augmentable resource, so both the host and CSD can collaborate in executing user applications. **6**- There should be an operating system running inside the CSD. This OS should provide a flexible environment to run a vast spectrum of applications in-place. **7**- The desired CSD should support distributed processing platforms such as Hadoop and MPI. **8**- The CSD should have the potential to implement ASIC- or FPGA-based accelerator engines to run highly demanding applications in-place with a compelling performance.

This chapter is composed of three sections. In the first section, we will provide a background on the modern SSD architectures and how ISP technology can be embedded in the SSD architectures. In the second and third sections, we will propose two CSD architectures, CompStor and Catalina, and show the path that led us to proposing a computational storage platform that can satisfy all the design goals mentioned above. In contrast to some of the related works that used simulation and modeling to propose an ISP architecture, in this research, a practical approach is used. We designed and prototyped two CSDs, namely CompStor and Catalina. Both of these CSD prototypes are fully functional and able to run a wide spectrum of user applications in-place. In this chapter, we will describe the hardware and software architectures of these two CSDs and discuss their strengths and weaknesses. At the end of this chapter, we will show why Catalina is not only an efficient CSD architecture but also a platform for implementing different ISP ideas and concepts.

3.1 Background

The storage system, where data originally reside, plays a crucial role in the performance of applications. In a system with multiple processing nodes, the data should be read from the storage units to the memory units of the application servers to be processed. As the

size of the data increases, the role of the storage subsystem becomes more important since the nodes need to talk to the storage units more frequently to fetch data and write back the results. Recently, data center architects have been considering SSDs over HDDs as the major storage units in the modern systems due to the former's better power efficiency and higher data transition rate [34].

SSDs use Nand flash memory as the storage media [35]. Nand memory chips are faster and more power-efficient than the magnetic disks that are used in HDDs, so SSDs are considered more efficient than HDDs. However, this advantage comes with complexity in the design and implementation of SSDs, wherein a multi-core controller is needed to manage the flash memory. Nonetheless, SSDs usually provide a high-speed interface to communicate with the host, such as NVMe over PCIe [12]. Implementing such an interface requires embedding more processing horsepower inside SSDs. In this section, we will first review the modern SSD architecture and explain the NVMe over PCIe, which is the host interface protocol of the proposed CSD architectures. Then, we will explain why an ISP-enabled storage unit could improve the performance and efficiency of the systems.

3.1.1 The SSD Architecture

The structure of a flash memory chip is shown in Fig. 3.1. A Nand flash memory chip is a package containing multiple dies. A die is the smallest unit of flash memory that can independently execute I/O commands and report status. Each die is composed of a few planes, and each plane contains multiple blocks. Erasing is performed in the block-level, so a flash block is the smallest unit that can be erased. Inside each block, there are several pages, which are the smallest units that can be programmed and written. The key point in this hierarchical architecture is the programmable unit versus the erasable unit. The Nand flash memory can be programmed in the page-level, which is usually 4–16 KB, while the

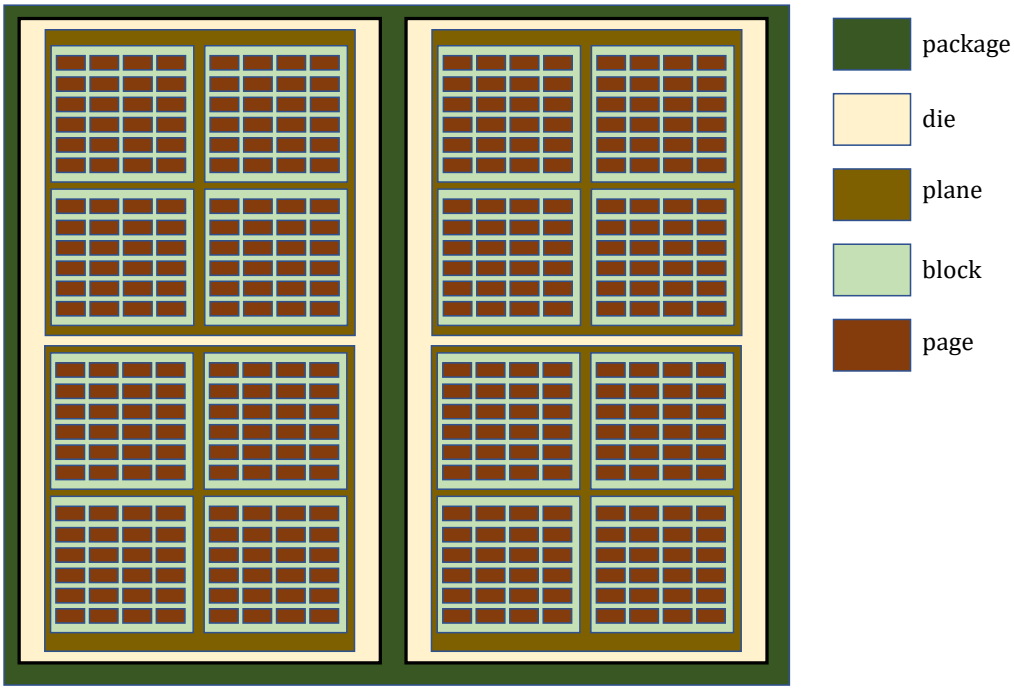


Figure 3.1: Flash chip organization

erase operation cannot be done on a smaller segment than a block, which is a few megabytes of memory.

The data cannot be simply overwritten on flash memory and should only be written on the erased blocks. This means if a page within a block should be updated, the SSD controller has to read the whole block's data, update the page content, and write back the data to a fresh and erased block. To modify the write operations, a garbage collector routine runs inside the SSD controller to erase the blocks during off-peak times to maintain optimal write speeds.

However, flash blocks can only be erased for a finite number of times, and they wear out as erase operations take place, so it is important to balance the number of erase operations among all the flash blocks of an SSD to increase the lifespan of the drive [36]. The process of leveling the number of erase operations is called wear leveling. In addition, the logical address space exposed to the host is different from physical block addresses, so there are multiple

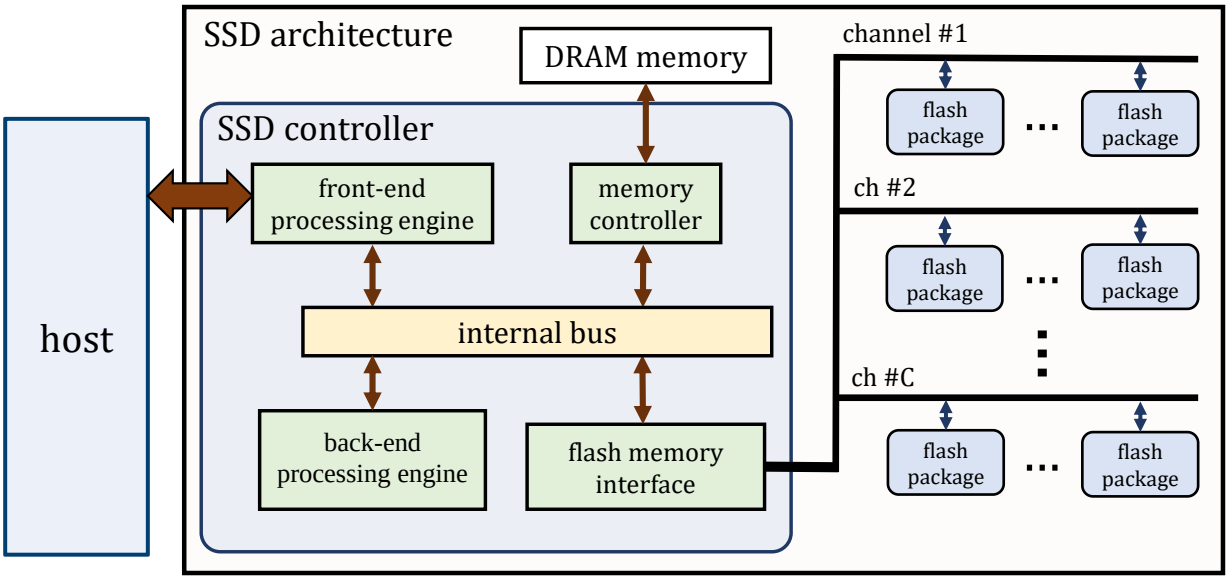


Figure 3.2: High-level overview of a modern SSD

tables for logical and physical address translation. The flash translation layer (FTL) is composed of all the routines needed to manage flash memory arrays, such as logical block mapping, wear leveling, and garbage collection.

An overview of the organization of an SSD is shown in Fig. 3.2. This figure demonstrates the modules that compose an SSD and how they collaborate to execute the host's I/O commands. These modules include the SSD controller, DRAM memory, and flash memory packages. The SSD controller talks to the host, receives the I/O commands, and controls the flash memory packages to serve the host. The DRAM memory is attached to the controller to be used by the controller's firmware routines. The flash packages are organized in channels, and all the channels can transfer data to the controller simultaneously. The number of channels and the bandwidth of each channel define the maximum bandwidth available to the internal components of the SSD.

There are four main components inside the SSD controller, namely the front-end processing engine, the back-end processing engine, the flash memory interface, and the memory controller. The front-end processing engine is responsible for communicating with the host via

protocols such as SATA, SAS, or NVMe over PCIe. It receives the I/O commands, checks their integrity, interprets them, and forward the commands to the back-end processing engine. The back-end engine handles the FTL, which includes garbage collection, physical and virtual address translation, error correction, and wear leveling routines. Note that there are other components, such as error correction unit (ECC), that are not shown in this figure for the sake of simplicity.

Super-scale data center designers have been trying to develop storage architectures that favor high-capacity hosts, and this fact is highlighted at Open Compute Summit (OCP) by Microsoft Azure and Facebook, which call for up to 64 SSDs attached to a host [37]. In Fig. 3.3, such a storage system is shown, wherein 64 NVMe SSDs are attached to a host via a PCIe switch. Modern SSDs usually contain 16 or more flash memory channels that can be utilized concurrently for flash array I/O operations. Considering 512 MBps bandwidth per channel, the internal bandwidth of an SSD with 16 flash memory channels is 8 GBps. However, the leading SSDs' specifications show that the host bandwidth is limited to about 1 GBps for random reads due to the complexity of the host interface software and hardware architecture [38, 39]. In other words, the accumulated bandwidth of all internal channels of the 64 SSDs reaches the result of the multiplication of the number of SSDs, the number of channels per SSD, and the bandwidth of each channel, which is equal to 512 GBps. Meanwhile, the accumulated bandwidth of the SSDs' external interfaces is equal to 64 multiplied by 1 GBps (the host interface bandwidth of each SSD), which is 64GBps.

Overall, there is an 8x gap between the accumulated internal bandwidth of all SSDs and the bandwidth available to the host. In other words, to read 32 TB of data, the host needs more than 8 minutes, while internal components of the SSDs can read the same amount of data in about 1 minute. Additionally, in such a storage system, data need to continuously move through a complex hardware and software stack between the host and storage interfaces, which imposes a considerable amount of energy consumption and dramatically

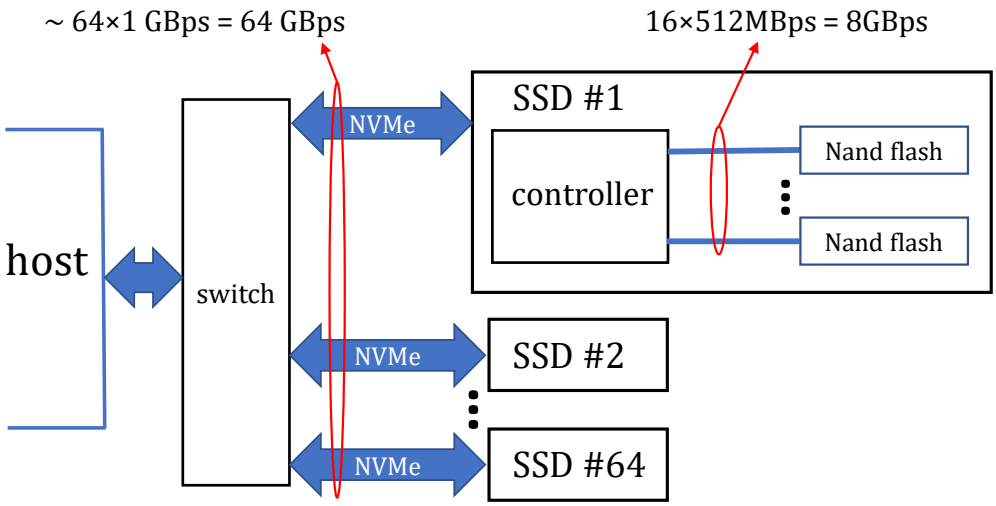


Figure 3.3: Storage systems' I/O bottleneck

decreases the energy efficiency of large data centers. Thus, storage architects need to develop techniques to decrease data movement; ISP technology has been introduced to overcome the data bottleneck challenge by bringing the process closer to the data.

NVMe: A High-Performance Interface for Non-Volatile Storage

There are different protocols to transfer data between the host and storage systems, such as SATA [10], SAS [11], and NVMe over PCIe [12]. The SSDs can execute multiple I/O commands simultaneously with low latency, and this feature is highlighted by some papers in the literature. Elyasi et al. improved the performance of a large-scale graph processing algorithm by 2x when they modified the algorithm based on the SSD architecture [40].

Among the aforementioned host interface protocols, the NVMe is proposed for SSDs and has an impressive performance in sending data between SSDs and hosts. Thus, the CSD architectures proposed by this research utilize this protocol. Although implementing the NVMe over PCIe protocol in a CSD is quite time-consuming, it is not within the scope of the contributions of this research. Thus, in this subsection, we will provide an overview of the NVMe protocol.

The peripheral component interconnect express (PCIe) [41] is a high-speed bus standard that is developed using a set of unidirectional pairs of serial and point-to-point links that are called lanes. A PCIe slot can have 1, 4, 8, or 16 lanes, which are shown by x1, x4, x8, and x16 notations, respectively. The PCIe is composed of three layers, namely the transaction layer, data link layer, and physical layer, and currently, there are four generations of the PCIe bus. Each lane of PCIe Gen1 provides 250 MBps of data bandwidth; Gen2 provides 500 MBps, Gen3 provides 985 MBps, and Gen4 provides 1970 MBps. This link can be used for connecting different peripherals to hosts, such as video cards, expansion cards, and storage units. The proposed CSD architectures in this research contain a host interface based on PCIe Gen3 x4, which can provide up to 3940 MBps of bandwidth.

NVMe protocol uses the PCIe data link to transfer data between a host and an SSD. In contrast to some traditional data transfer protocols developed around the HDDs that do not have more than a queue for submitting I/O commands, NVMe protocol is designed based on the SSD architectures. In other words, since SSDs can run multiple I/O commands at the same time, NVMe is designed to take advantage of this feature, and it provides up to 64K data transition queues, while each queue supports up to 64K parallel I/O commands.

NVMe over PCIe protocol is developed based on the non-uniform memory access (NUMA) model [42, 43]. In this model, two systems that are connected can access each other's memory; however, the character of the local memory access is different than the character of the remote memory access. In other words, this model provides high-performance memory access for the host and the SSD.

3.1.2 ISP: Bring the Process to Data

In a traditional CPU-centric scheme, data always move from storage devices to processing engines. This mechanism, which is inherently limited by the *von Neumann bottleneck*, is the

root cause of the data bottleneck challenges mentioned in the previous subsection, especially when many SSDs are connected to a host. ISP technology proposes a contrary approach to push the concept of "bring the process to data" to its ultimate boundaries wherein processing engines inside storage units take advantage of internal high-bandwidth, low-power data links and process data in-place. In fact, "bring the process to data" is the same concept that led to the emergence of distributed processing platforms such as Hadoop and Spark [44]. Later in this dissertation, we will discuss how the Hadoop platform and ISP technology can simultaneously work together in a cluster.

The ISP technology minimizes the data movements between the host and storage units and also increases the processing horsepower of a system by augmenting energy-efficient processing engines to the whole system. This technology can potentially be applied to both HDDs and SSDs; however, modern SSD architecture provides a better environment for developing this technology. SSDs that can run user applications in-place are called CSDs. These storage units are augmentable processing resources, which means they are not designed to replace the high-end processors of modern servers. Instead, they can collaborate with the host's CPU and augment their efficient processing horsepower to the system.

It is noteworthy that CSDs are fundamentally different than object-based storage systems such as Seagate Kinetic HDDs [45], which transfer data at the object-level instead of the block-level. The object-based storage units can receive objects (e.g., images) from a host, store them, and, at a later time, retrieve the objects back to the host using object identifications. Consequently, the host's filesystem does not need to maintain metadata of block addresses of the objects. On the other hand, CSDs can run user applications in-place without sending data to a host.

3.2 CompStor: The First Linux-Powered CSD

The first CSD that we designed and prototyped is called CompStor [46], which stands for "computational storage." CompStor is the first computational storage with a dedicated quad-core application processor as the ISP engine, which runs a full-fledged Linux operating system. This ISP engine is revolutionary enough to make CompStor a very flexible CSD compared to similar works that struggle to run ISP tasks on real-time processors or FPGA-based accelerators. This section includes three subsections to describe the CompStor hardware architecture and software stack as well as the CompStor prototype and experimental results.

3.2.1 Hardware Architecture

Since the development of the firmware of a regular SSD is time-consuming and error-prone, for the first CSD design, we chose to separate the development of the conventional flash management functionalities from the innovative ISP capabilities to avoid potential errors and extra complications. Therefore, CompStor is composed of two boards that implement the flash management routines and ISP engine. Fig. 3.4 demonstrates a high-level overview of the CompStor hardware architecture. This architecture is composed of two separate parts, namely the *conventional subsystem* and *ISP engine*.

The *conventional subsystem* contains a controller, 2 GB DRAM memory, and an array of flash packages. The controller is composed of two MicroBlaze processors [47], an ECC unit, a host NVMe over PCI interface, a memory controller, and a flash memory interface. All of these components are designed and implemented in the FPGA. Among these components, the two MicroBlaze processors are the front-end and back-end processors that run the SSD controller firmware and control the other modules.

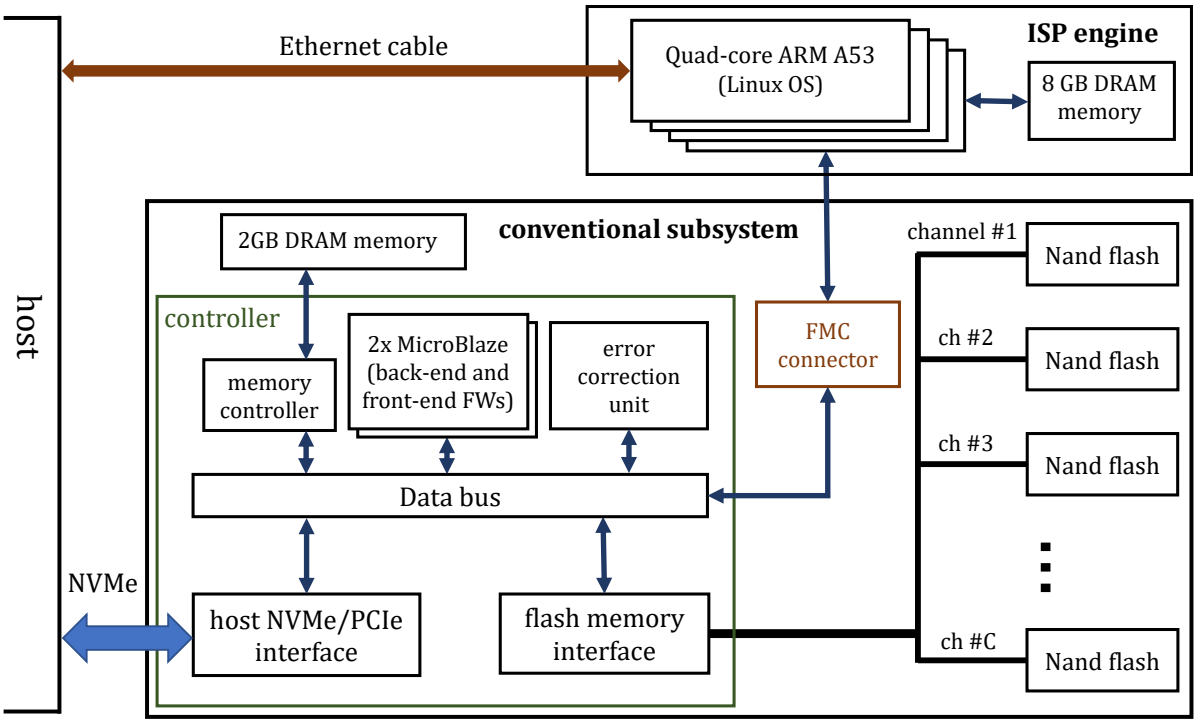


Figure 3.4: The CompStor hardware architecture

Data stored in the flash packages may change due to the transient errors that usually happen in memory cells. The ECC unit takes care of these errors by utilizing error-correction algorithms such as low-density parity-check (LDPC) [48]. The host interface is responsible for communicating with the host via the NVMe over PCIe protocol. It receives the I/O commands, checks their integrity, and sends them to the front-end processor. The front-end processor interprets the commands, performs internal DRAM initializations, and then asks the back-end processor to execute the host's commands. The back-end processor controls the flash memory interface to communicate with the flash packages and fulfills the host's commands. After the back-end processor finishes the I/O operations, a completion command is sent to the front-end processor, and, accordingly, the host is notified that the I/O command is finished.

There is an internal data bus in the conventional subsystem that transfers data between different components. This bus is where we can attach the innovative ISP engine, which is responsible for running user applications. In other words, the ISP engine is attached as an

external utility to the conventional subsystem to augment the ISP capabilities to the storage unit. The ISP engine includes a quad-core ARM Cortex-A53 processor [49] as well as 8 GB of dedicated DRAM memory. There is an FPGA mezzanine card (FMC) connector [50] that forms the connection between these two subsystems. In addition, an ethernet connection has been provided in CompStor to allow for a TCP/IP connection between the ISP engine and the applications that run on the host.

3.2.2 CompStor Software Stack

In CompStor, a host-side *client* application controls the ISP flow, so from a master-slave perspective, the *client* is the master, while CompStor behaves as the slave. The *client* needs to perform a defined sequence of steps: sending an ISP task to CompStor, waiting for the completion of the task, and receiving the results of the execution. In this subsection, the software stack that helps the user go through these steps will be discussed.

We implemented an ad-hoc messaging protocol for ISP-related data transfer between the *client* running on the host and the ISP engine in CompStor via the Ethernet cable (see Fig. 3.4). This ISP messaging protocol includes different types of messages. In fact, they are virtual entities traveling through layers of the software stack to deliver ISP-related information and may get encapsulated into other messages. Each layer may either process or redirect them to the next layer. The different message entities in the CompStor software stack are as follows:

1. ***Command*:** A data structure containing detailed information about ISP tasks, including the name of input and output files, the Linux shell command/script or the executable file name, the arguments needed to pass to the application, and the access permissions.

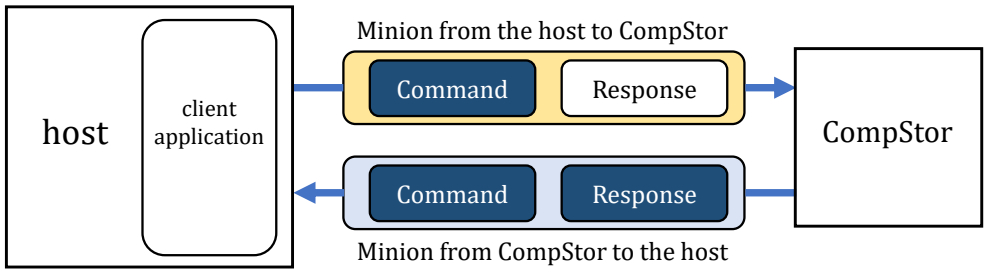


Figure 3.5: A *Minion* message travelling between host and CompStor

2. ***Response*:** A data structure containing the information about the outcome of an ISP task, such as the final status of the *command* and the time consumed to execute the ISP task inside CompStor.

3. ***Minion*:** A virtual entity that travels from a *client* to CompStor and delivers a *command*. It waits until the ISP task is finished to deliver the *response* back to the *client*. This virtual entity is composed of a *command* and a *response*. The *command* part is populated by the *client*, while the *response* is populated by CompStor. Fig. 3.5 depicts a *minion* containing a *command* and a *response* traveling between a *client* and CompStor.

4. ***Query* :** A virtual entity that travels from a *client* to CompStor to deliver an administrative message. Similar to a *minion*, it travels back to the *client* after delivering the message, but it cannot trigger an ISP task. Instead, it can load an executable to the ISP engine or obtain information about the current status of CompStor such as processor utilization and temperature.

The software stack consists of multiple layers spread over the host and CompStor. Each layer is responsible for a specific task and serves other layers. The *commands*, *responses*, *minions*, and *queries* are the only entities traveling from one layer to another. These layers are defined as follows:

- **Client:** An administrative C/C++ application that controls the ISP flow, i.e., it sends

minions to CompStor and waits for the result of the ISP task.

- **ISP executable:** A C/C++ application, a Linux shell command/script, or a combination of both that the user desires to run inside CompStor.

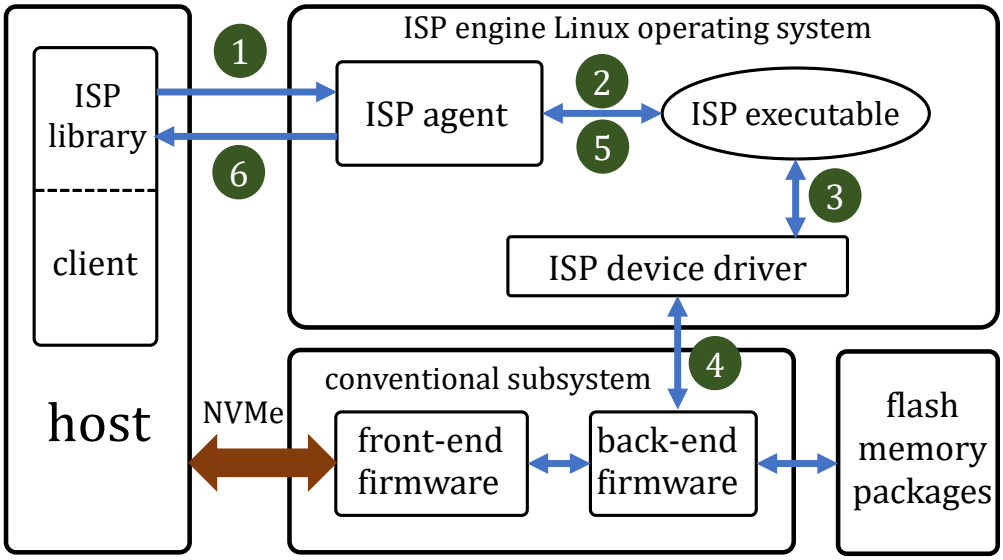
- **ISP library:** A C/C++ library that provides high-level APIs for the *client* administrative application to control the ISP flow. The ISP library is intended to be used in the *client*, not in the *ISP executable*. This means the CompStor software stack is invisible to the *ISP executable*, and the user can reuse applications that are developed for running in regular hosts.

- **ISP agent:** A daemon that runs in the ISP engine and is responsible for receiving *minions* from the *client* and spawning ISP applications in CompStor. After ISP command completion, the daemon populates the *response* field of the *minion* and sends it back to the *client*.

- **ISP device driver:** A Linux device driver that is implemented in the kernel space of the CompStor Linux operating system and communicates to the conventional subsystem for flash data access. This device driver abstracts the flash read/write accesses, so the *ISP executable* can read and write to flash memory similar to when it runs in a host.

- **Conventional subsystem's back-end firmware:** The firmware that is responsible for flash management tasks and implements FTL. This firmware runs in the conventional subsystem and talks to the *ISP device driver* via the FMC connector to transfer data between the conventional subsystem and the ISP engine.

Fig. 3.6 depicts the software stack architecture and shows how the layers communicate with each other to accomplish an end-to-end ISP process. When the *client* launches a *minion*, it triggers multiple message transfers between different software layers. Fig. 3.6 also describes



Step #	Description
1	Host side *client* configures a *minion* and sends it to the *ISP agent* using the *ISP library* APIs.
2	The *ISP agent* extracts *command* from the received *minion* and spawns the *ISP executable*.
3	At runtime, the *ISP executable* accesses the flash storage via the *ISP device driver*.
4	The *ISP device driver* sends read/write commands to the *back-end firmware* that handles flash management routines.
5	At runtime, the *ISP agent* keeps track of the progress of the *ISP executable*.
6	In the end, the *ISP agent* populates the *response* field in the *minion* and sends the *minion* back to the *client*.

Figure 3.6: CompStor software stack and the step-by-step description of the ISP flow

the lifespan of a *minion*, from the time it is configured in *client* to when it delivers the result back to the *client*. The *client* is able to send several concurrent *minions* to multiple CompStor CSDs attached to a host. This gives the *client* the ability to trigger parallel ISP applications.

3.2.3 Prototype and Experimental Results

In this subsection, we demonstrate a fully functional CompStor prototype and run several experiments to investigate the energy consumption and performance of running I/O- and compute-intensive applications using CompStor CSDs. The prototype is an NVMe CSD with an FPGA-based controller coupled with an ISP engine built around a quad-core 64-bit ARM Cortex-A53 application processor.

Fig. 3.7 shows the prototype that was developed for running the experiments. For the prototype, we built two boards, which were completely aligned with the hardware architecture described in Fig. 3.4. In fact, the whole CSD controller was implemented using an ISP engine attached to the conventional subsystem via an FMC connector. For the implementation of the conventional subsystem, we used a Xilinx Vertex-7 2000T FPGA, while the ISP engine was implemented using a Xilinx Zynq Ultrascale plus MPSoC. The latter is an MPSoC chip containing an FPGA together with a quad-core 64-bit ARM Cortex-A53 processor. This MPSoC also contains two ARM Cortex-R5 real-time processors [51], a GPU, and a set of ASIC modules such as encryption and decryption units. However, we did not use these utilities and modules in the CompStor. To the best of our knowledge, the CompStor prototype is the first Linux-powered CSD equipped with a software stack to support running user applications in-place seamlessly.

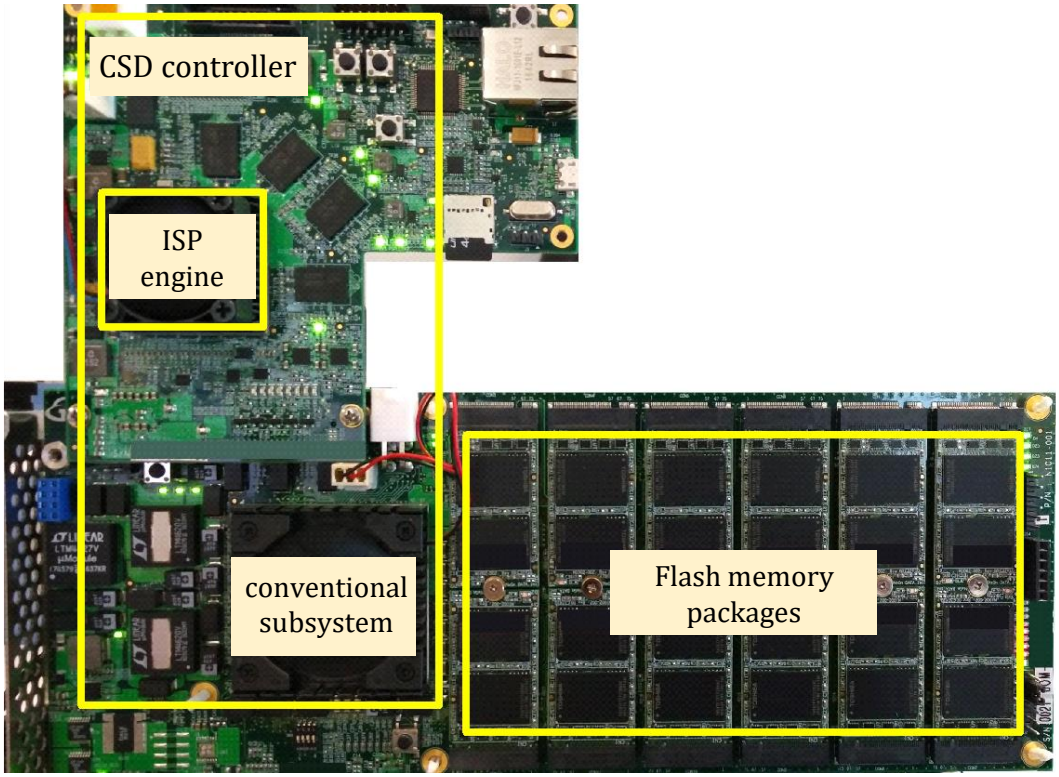


Figure 3.7: CompStor prototype

Experimental Setup

To run the experiments, we built a host with 16 CompStor CSDs (see specifications in Table 3.1). Since processing very large text files is common in super-scale applications, for the experiments described in this subsection, we prepared a dataset that contained 348 compressed big text files selected from the Gutenberg dataset [52]. These text files were books written by different authors that were transformed into plain text files. The total size of the dataset was about 11.3 GB. The definition of performance in this subsection is the amount of data that is processed in a time unit (second), and energy consumption is defined as the amount of energy the system needs to process one gigabyte of data.

The application set selected for the experiments included both I/O- and compute-intensive applications. Compression and decompression algorithms are commonly used in super-scale applications. Thus, we used gzip/gunzip [53] and bzip2/bunzip2 [54] algorithms as the

Table 3.1: The experimental server specification

CPU type	Intel Xeon E5-2620 v4
memory	32 GB DDR4
operating system	Ubuntu 16.04
conventional SSDs	4x 256GB NVMe SSD
computational storage device	16x CompStor NVMe CSD

compute-intensive applications. For the I/O-intensive experiments, two search applications were selected, namely grep [55] and gawk [56]. Grep is a standard Linux shell command designed to search in text inputs, while the gawk utility searches text and makes changes based on user-specified patterns.

Experimental Results: Performance

In the first set of experiments, we used a host-side *client* to trigger ISP tasks on CompStor CSDs. We ran the applications mentioned above using different numbers of CompStor CSDs to show the scalability of the proposed solution. Expectedly, the performance of one CompStor was lower than a high-end Intel Xeon processor; however, the performance improvement scaled as the number of CSDs increased. Fig. 3.8 depicts how performance scaled with different numbers of CompStor CSDs.

Even though the aggregated performance of multiple CompStor CSDs can equal or surpass that of a high-end x86-based processor, it makes sense to consider that one augments the other and results in higher performance and a more efficient system. Fig. 3.9 depicts the performance of the Xeon processor combined with the performance of multiple CompStor devices when running the bzip2 compression algorithm. In this experiment, we distributed the input files between the host and CompStor CSDs; then, the performances of the CompStor

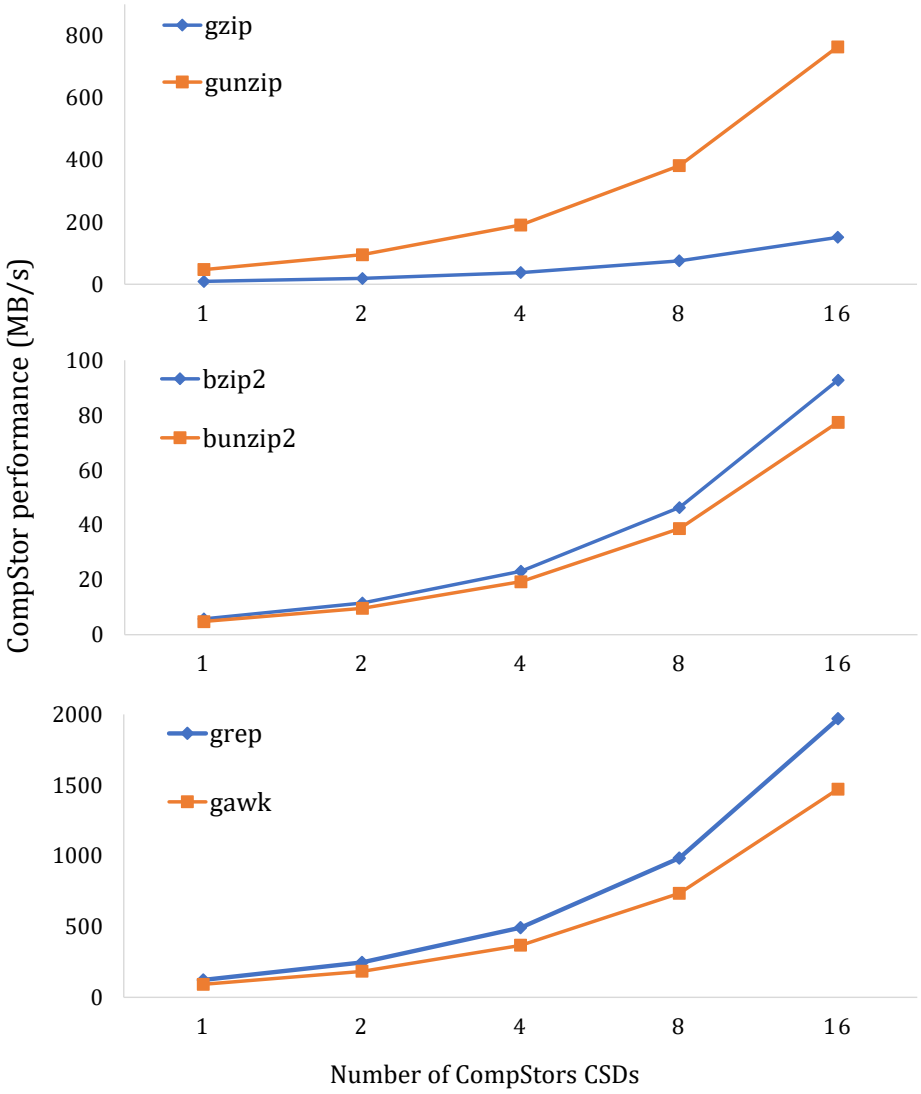


Figure 3.8: CompStor performance for running I/O- and compute-intensive applications

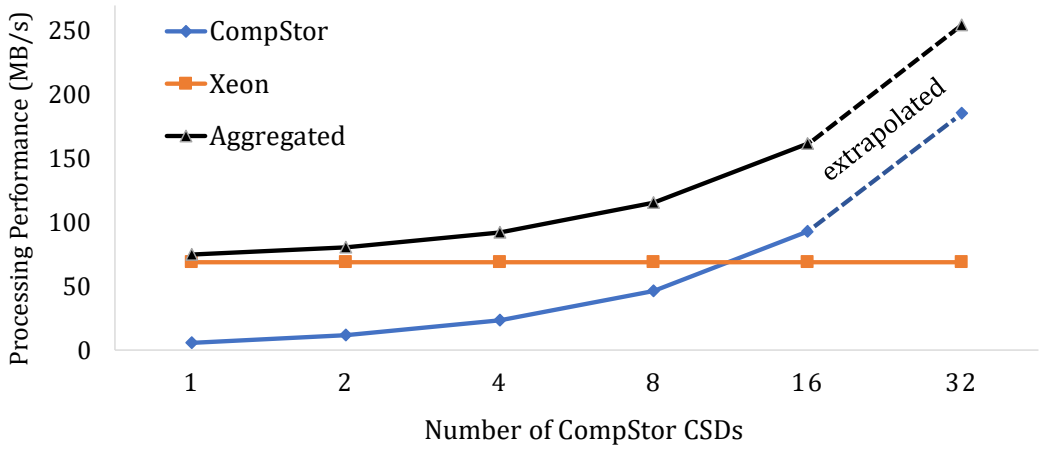


Figure 3.9: Host's CPU and CompStor CSDs aggregated performance for running bzip2

CSDs and the host are measured. These results show that the CompStor CSDs augmented a considerable processing horsepower to the whole system and were flexible enough to run different I/O- and compute-intensive applications.

Experimental Results: Energy

In this experiment, we have utilized 16 CompStor CSDs for running different applications and measured the energy efficiency of the developed system for comparing the deployment of the regular SSDs versus the CompStor CSDs. In this experiment, we utilized 16 CompStor CSDs to run different applications and measured the energy efficiency of the developed system to compare the deployment of the regular SSDs versus the CompStor CSDs. In the latter case, for the computation to take place, only the ISP *minions* needed to be transferred between the host and the CSDs, greatly reducing the interface traffic and required energy. The reason we chose energy consumption over power consumption was to make the results of these experiments independent of the performance of the system. We ran the experiments and measured the energy consumed when executing compression, decompression, and search applications. The results were normalized per gigabyte of data processed, i.e., J/GB, as shown in Fig. 3.10. The experimental results show up to 3.3x improvement in energy

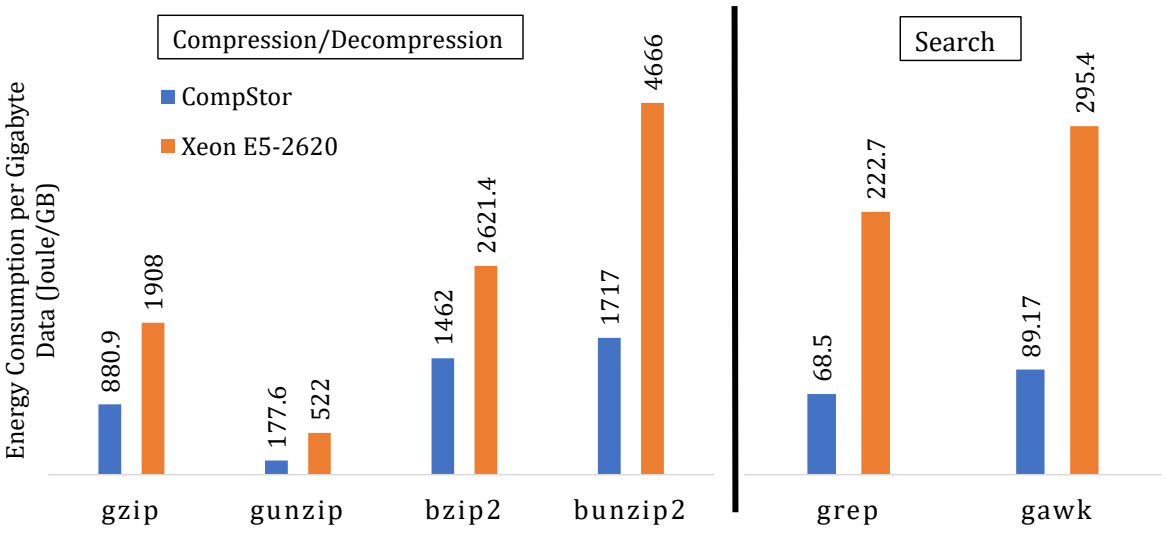


Figure 3.10: Conventional SSDs versus CompStor CSDs energy consumption

efficiency in comparison to the host CPU utilizing conventional SSDs.

CompStor shows good flexibility to run different I/O- and compute-intensive applications and can also improve the performance and energy efficiency of a system considerably. However, there is an important problem with CompStor's design. CompStor is introduced as an ISP device, yet data still need to move from the conventional sub-system to the ISP engine. Although this data transfer is less expensive than data transfer via a complex NVMe over PCIe interface, this off-chip data transfer is not aligned with the core ISP definitions. The off-chip data transfer between the conventional subsystem and the ISP engine increases the latency and also imposes energy consumption on the whole system. In other words, any application run on the ISP engine will suffer from the off-chip data transfer latency and energy consumption. To solve this problem, we introduced Catalina, which is an MPSoC-based computational storage platform with potentials to implement different ISP engines and run a wide spectrum of applications. In the next section, we will describe the software and hardware architectures of Catalina.

3.3 Catalina: An SoC-based ISP Platform

In this section, we will describe the hardware and software architectures of Catalina [57], which propose to satisfy all the design goals mentioned earlier in this chapter. Unlike CompStor, the Catalina controller contains all the conventional flash management components and the ISP engine implemented on a single chip, so data do not need to be transferred off-chip for ISP. This section is composed of three subsections. In the first subsection, different hardware components of Catalina will be described, and we will discuss how they work together. The second subsection will define the Catalina software layers that make it possible to send ISP commands, process data in-place, and write the results back to flash memory. Finally, the third subsection will demonstrate the fully functional Catalina prototype, which was used to investigate the benefits of deploying CSDs in clusters.

3.3.1 Hardware Architecture

Catalina was developed based on the Xilinx Zynq Ultrascale plus MPSoC chip [58]. This device is composed of two subsystems, namely programmable logic (PL) and a processing system (PS). The PS is an ASIC-based processing subsystem that includes a quad-core ARM Cortex-A53 64-bit processor equipped with Neon SIMD engines and floating-point units, two ARM Cortex-R5 real-time processors, a DRAM controller, and other interconnect and data movement components. Adjacent to the PS is the PL subsystem, which is an FPGA that can be utilized to implement different components of the CSD controller, such as the host and flash memory interfaces. These two subsystems are packaged in one chip, with multiple data links connecting them for high-performance, power-efficient intra-chip data transfers. Together, these two subsystems provide a suitable platform for implementing conventional SSD routines as well as running user applications in-place.

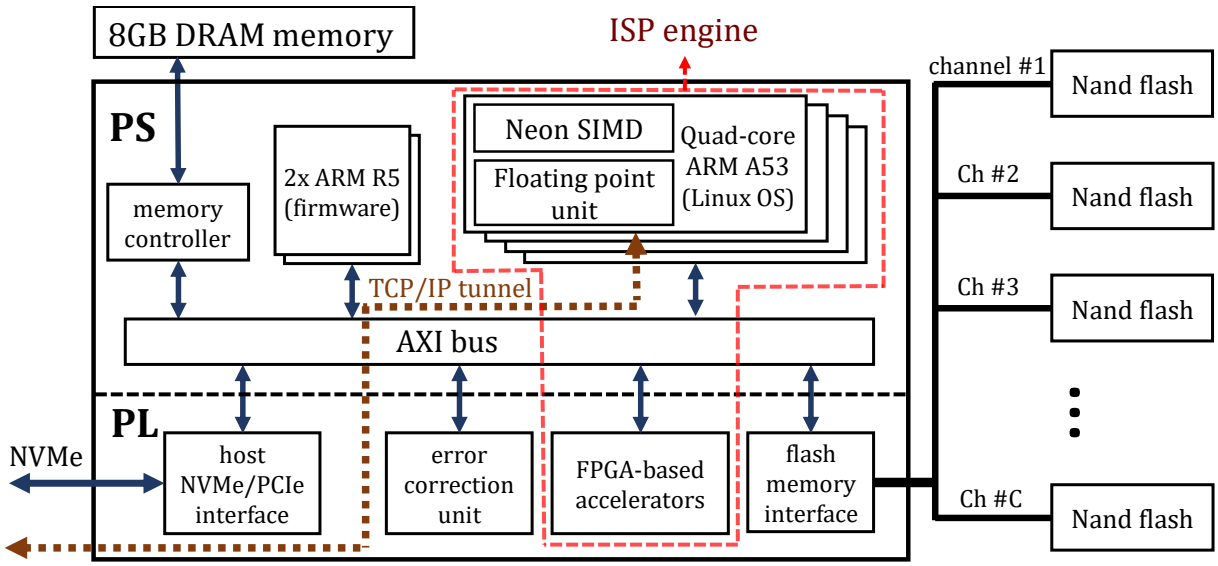


Figure 3.11: Catalina hardware architecture

Fig. 3.11 shows the Catalina architecture implemented using the Xilinx Zynq Ultrascale plus MPSoC chip. On the PL subsystem, there are three conventional components of the controller: the host NVMe over PCIe interface, the ECC unit, and the flash memory interface. The host interface is responsible for sending and receiving the NVMe commands from the host and checking the integrity of the commands. The ECC unit enables the controller to correct the data errors that regularly occur in the flash memory array. The flash memory interface communicates with the flash memory channels. In Fig. 3.11, each flash channel is connected to a set of flash memory packages, and the flash memory interface talks to the flash packages on all the channels concurrently.

On the PS, there are two ARM Cortex-R5 real-time processors that are used for both controlling the components implemented in the PL as well as running the FTL routines. In fact, the conventional firmware routines run on these two real-time processors. Similar to CompStor, Catalina has two firmware processors, front-end and back-end. However, instead of the MicroBlaze processors that are used in the CompStor design, Catalina uses the ARM Cortex-R5 processors of the Xilinx Zynq Ultrascale plus MPSoC chip. In other words, one of the real-time processors runs the front-end (FE) firmware, which controls the host interface

module and interprets the host's I/O commands, while the other ARM Cortex-R5 processor runs the back-end (BE) firmware, which is responsible for controlling the error correction and flash interface units. The BE firmware also runs other essential FTL routines, such as garbage collection and wear leveling.

All of the components mentioned above are common among conventional SSDs; however, Catalina is equipped with a unique ISP engine. This engine is dedicated to running user applications in-place, and it contains a quad-core ARM Cortex-A53 processor equipped with Neon SIMD engines and floating-point units (FPUs). The quad-core processor is capable of running a vast spectrum of applications, while the Neon SIMD engines can increase the performance of some compute-intensive applications. Overall, the quad-core Cortex-A53 processor is the main ISP engine of Catalina, and the Neon SIMD engines as well as the FPUs can accelerate user applications running in-place. Additionally, since there is an FPGA on the PL subsystem, it is possible to implement FPGA-based accelerators to boost the performance of specific applications significantly.

As Fig. 3.11 demonstrates, both the ISP engine and the two Cortex-R5 real-time processors which run the conventional flash management routines are packaged in the same chip. These two engines are connected via an internal ARM advanced extensible interface (AXI) bus [59]. The shared AXI bus makes it possible to transfer data between the BE firmware and the ISP engine efficiently. In other words, the ISP engine can bypass the whole NVMe hardware and software stack and access the data stored in the flash memory array directly by communicating with the BE firmware. There is also an 8 GB DRAM memory connected to the AXI bus that is shared among all the processing units. This shared memory, which is not available in the CompStor architecture, can be used for data transfers between the conventional subsystem and the ISP engine.

3.3.2 Catalina Software Stack

The most important part of the software components is the operating system running inside the ISP engine. Thus, similar to CompStor, we ported a full-fledged Linux OS running on the quad-core ARM Cortex-A53 processor. This OS provides a flexible environment for both running user applications in-place as well as implementing other layers of the software stack. 3.12 demonstrates the architecture of the software layers and how they make it possible to run applications in-place.

In Fig. 3.12, there is a cluster of M hosts connected using a TCP/IP interconnect, and the host #1 is attached to the N Catalina CSDs via a PCIe switch. In this figure, the lowest layer of the software stack is the BE firmware, which implements the FTL procedures. The BE firmware serves both the FE firmware, which talks to the host via NVMe protocol, as well as a block device driver implemented in the kernel space of the ISP engine's operating system. The block device driver issues flash I/O commands directly to the BE firmware, so the data link through the block device driver bypasses the NVMe over PCIe software and hardware stack. The block device driver also makes it possible to mount the flash storage inside the Catalina OS. In other words, any user application that runs in-place has filesystem-level access to the data stored in the flash memory array via a high-performance, low-power internal data link.

However, the ISP engine should also provide a link between applications that run in-place and applications on the host. Thus, in addition to the block device driver, we implemented a TCP/IP tunnel through NVMe protocol to transfer TCP/IP packets between the applications running on the host and the applications inside Catalina. Such a link was not available in CompStor, and we had to use an Ethernet cable to talk to the host. In Catalina, we utilized NVMe vendor-specific commands to packetize TCP/IP payloads inside the NVMe commands (the TCP/IP tunnels through NVMe are demonstrated in Fig. 3.12 by dashed

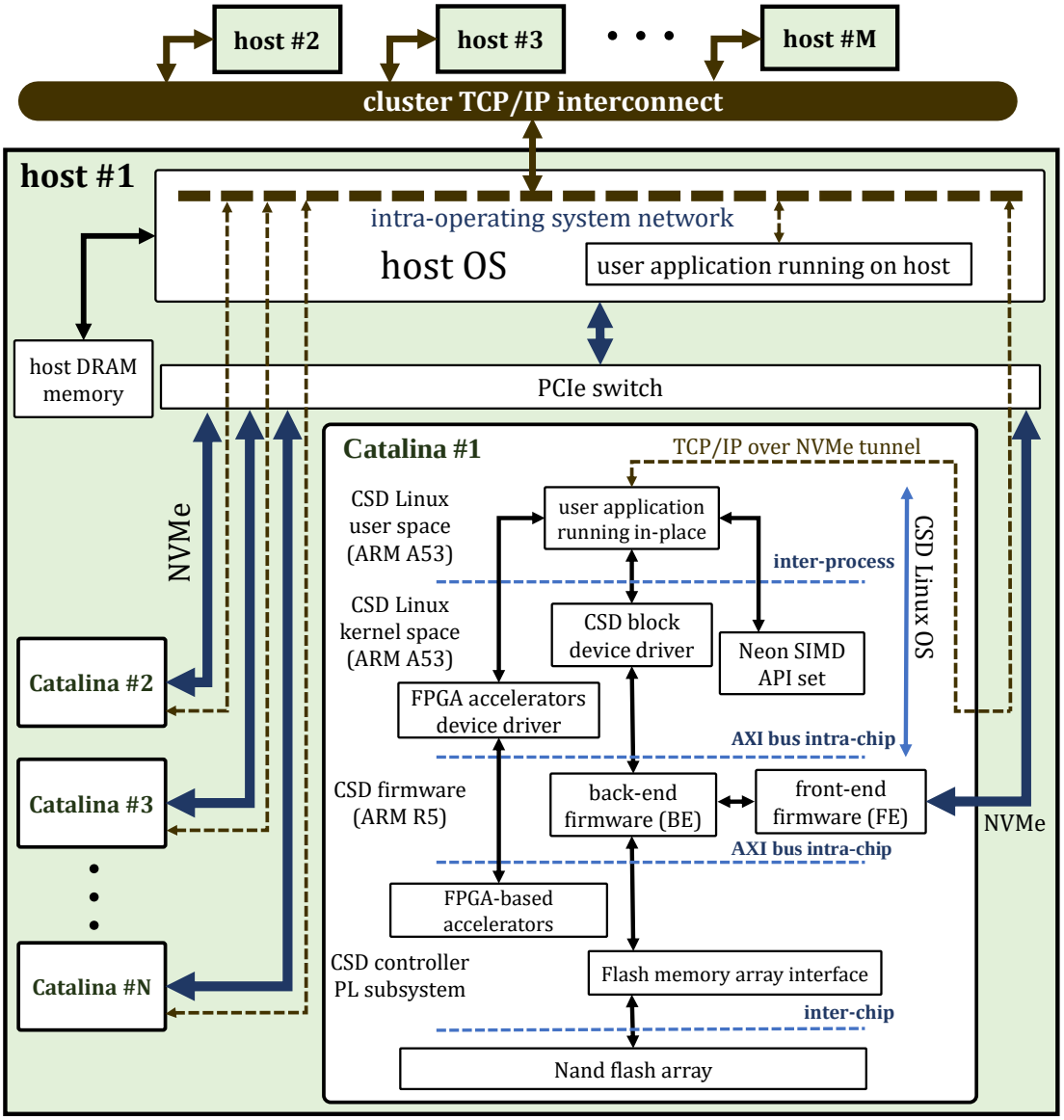


Figure 3.12: Catalina software stack

lines). A software layer implemented on both host OS and the Catalina OS provides the tunneling functionality. Since distributed platforms such as Hadoop MapReduce and MPI are based on TCP/IP connection, this link plays a crucial role in running distributed applications. As shown in Fig. 3.12, all the N Catalina CSDs that are attached to the host #1 can concurrently communicate with applications running on the host.

It is noteworthy that by using Linux TCP/IP packet routing tools, we can create an internal network in the host operating system and reroute the packets sent or received by the Catalina CSDs to the other hosts attached to the TCP/IP interconnect (see Fig. 3.12). In other words, if several hosts are connected via a TCP/IP interconnect—each of them equipped with multiple Catalina CSDs—the hosts, as well as the CSDs can communicate with each other via a TCP/IP network. Such a CSD-equipped cluster architecture benefits from the efficient ISP capabilities of Catalina CSDs to run distributed applications. In fact, the proposed CSD architecture is an augmentable processing resource, which is adoptable in the cluster without any modifications in the underlying Hadoop or HPC platforms.

Additionally, the user applications that run in Catalina CSDs have access to Neon SIMD engines via a set of application programming interfaces (APIs) provided in the Catalina operating system. Using these APIs, user applications can potentially be accelerated by the Neon SIMD engines. Overall, user applications have access to four unique tools as follows: 1.) a high-speed, low-power internal link to the data stored in the flash memory, 2.) a TCP/IP link to the applications running on the host, 3.) a set of APIs to utilize the Neon SIMD engines, and 4.) the FPGA-based accelerators that can potentially be implemented in the PL subsystem.

The last layer of the software stack is the synchronization layer between the host and the Catalina operation systems. These two operating systems can access the data stored in the flash memory array at the filesystem-level and concurrently mount the same storage media, which is a problematic behavior without a synchronization mechanism. In CompStor, we

avoided such a problem by partitioning the flash memory into two parts: 1.) a partition that is accessible by the host, and 2.) another partition that is accessible by the ISP engine. In this case, the host could still access the ISP engine's partition by a mounting/unmounting mechanism; however, such a mechanism adds a delay when both the host and ISP engine need to access the same partition frequently.

In the Catalina software stack, to address the synchronization issue, we implemented the Oracle cluster filesystem 2nd version (OCFS2) [60] between the host and the CSD. Using the OCFS2, both the host and Catalina CSD can issue flash I/O commands and mount the shared flash memory natively. This is the main difference between the OCFS2 and network filesystem (NFS) [61]. In the NFS, only one node mounts the shared storage natively, and other nodes use a network connection to access the shared storage, so NFS limits the data throughput and also suffers from the *single point of failure* problem. Meanwhile, using the OCFS2, all nodes can mount the flash memory natively.

3.3.3 Catalina Prototype

To prove the feasibility of the proposed ISP solution and investigate the benefits of deploying Catalina CSDs in clusters, we designed and built a fully functional prototype of Catalina that completely aligns with the hardware and software architectures described in the previous subsections. Fig. 3.13 shows the Catalina CSD prototype. The CSD controller implemented on a Xilinx Zynq Ultrascale plus MPSoC as well as the Nand flash packages are shown in this figure.

The CSD controller is composed of the PS and PL subsystems, which implement the Catalina conventional and ISP engines. The hardware specifications of the Catalina prototype are in Table 3.2, separated for these two engines.

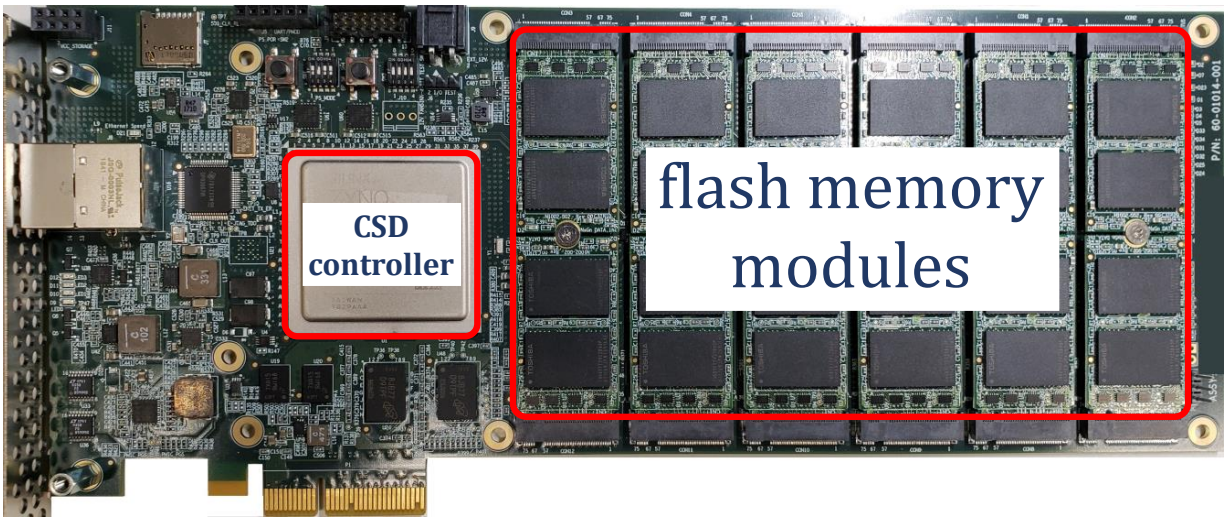


Figure 3.13: Catalina prototype

Table 3.2: Catalina prototype hardware specifications

conventional subsystem	processing units	2x ARM R5 processors @600MHz FPGA @250MHz
	host interface	NVMe over PCIe Gen3
ISP engine	processing units	1x quad-core ARM Cortex-A53 with Neon SIMD engines and FPUs FPGA-based accelerators
	host interface	TCP/IP tunnel over NVMe
shared	DRAM memory	8 GB

The prototype of Catalina is able to execute the host's I/O commands and also provides a user-friendly mechanism for offloading the applications to the CSD via a TCP/IP tunnel through a NVMe over PCIe link. Considering multiple Catalina prototypes attached to a host, an administrative application on the host can initiate parallel and distributed tasks on CSDs while the host and the CSDs' operating systems are synchronized by the OCFS2 filesystem.

The user applications that run inside the Catalina CSDs could potentially be developed in any language supported by Linux OS. In addition, since there is a TCP/IP tunnel to the host, ISP users can easily connect to the internet to extend the libraries and languages that are supported by the Linux operating system inside Catalina. The applications can interact with the flash memory at the filesystem-level, i.e., open files, process them, and write back the results similar to when they run on a conventional host.

Despite all the projected benefits of deploying the CSDs, they should be cost-effective to be adoptable in the clusters. After prototyping Catalina, a sensible cost analysis of manufacturing CSDs can be presented. Compared with a regular SSD based on a conventional controller, a CSD should be equipped with more processing horsepower to run applications in-place efficiently. Interestingly, according to our observations as well as the SSD bill of material analysis [62, 22], the difference between SSD and CSD manufacturing costs is insignificant, since the SSD manufacturing cost is largely dominated by the flash memory chips. The cost of flash memory chips is about 75% of the SSD price [63]. With other miscellaneous costs (such as DRAM, miscellaneous components, and manufacturing costs) that would account for 20-25% of the SSD price, the controller would account for, at most, 5% of the SSD price.

Overall, Catalina is a CSD that has a dedicated ISP engine with a quad-core application processor equipped with Neon SIMD engines and FPUs. There is a full-fledged operating system ported to the ISP engine that provides a flexible environment for running a wide

spectrum of applications. The user can also implement FPGA-based accelerators to boost the performance of applications. The applications running in the ISP engine can access the flash memory data at the filesystem-level through a highly-efficient intra-chip link, while the host and ISP engine operating systems are synchronized by the OCFS2 cluster filesystem. All these ISP features are available in Catalina without imposing a major overhead on the manufacturing cost.

Since Catalina with the features mentioned above has potentials for implementing different types of FPGA-based accelerators, utilizing ASIC-based processing engines, and being used in distributed environments such as Hadoop and MPI, it can be considered a computational storage platform. In the rest of this dissertation, we will use Catalina as a platform to implement various ISP ideas as well as investigate the benefits of deploying Catalina in different systems and applications.

Chapter 4

ISP-Enabled Distributed Platforms

Catalina was developed concerning a straightforward deployment in distributed environments. Since it has all the required features to play the role of a regular processing node, the system architects do not have to make major modifications in the underlying platforms for deployment of the Catalina CSDs. After attaching Catalina CSDs to a host and setting the network configurations, the CSDs are exposed to the other hosts in the cluster by their network addresses (e.g., IPs). In other words, from a system-level point of view, the Catalina CSDs are similar to regular processing nodes, and the underlying ISP hardware and software details are invisible to other nodes in the cluster. In this chapter, the first section will provide an overview and explain how Hadoop, MPI, and cluster filesystems work. In the second section, we will show how Catalina CSDs can be deployed in such clusters as well as investigate the benefits of ISP-enabled Hadoop and MPI-based clusters.

4.1 Background

A cluster is a set of nodes that work together to accomplish a distributed task. In a cluster, there is an interconnect network to provide connectivity among the nodes. The configuration of these nodes can be uniform so that all nodes are similar to each other, i.e., they use the same hardware and operating system, or they can have different configurations [64]. Overall, these nodes form an environment for running tasks in a distributed fashion. In this section, we will provide an overview of this class of platforms and the cluster filesystem.

4.1.1 Distributed Processing Platforms

A while ago, when the cost of data movement was insignificant in comparison to the computational cost, there could be a centralized storage system, and other hosts had to send I/O requests to fetch data. With this mechanism and today's volume of data, a data-intensive application requires large amounts of data to be fetched from the centralized storage system, and such huge data movements drastically increase energy consumption. With the emergence of big data, the storage system can no longer be centralized, and the centralized approaches come short of satisfying super-scale applications' demands, which call for scalable distributed processing platforms. To answer these demands, distributed processing platforms such as Hadoop have been proposed to process data near where they reside [65].

Hadoop has emerged as the leading computing platform for big data analytics and is the backbone of hyper-scale data centers [66], wherein hundreds to thousands of commodity servers are connected to provide service to clients. The Hadoop distributed processing platform consists of two main parts, namely the Hadoop filesystem (HDFS) [67] and MapReduce engine [68]. Hadoop was inspired by the Google filesystem (GFS) and Google publications related to the MapReduce programming model [69, 70].

The Three main characters of the Hadoop distributed platform are:

1. **Scalability:** A Hadoop cluster can be scaled up very well. This feature makes it capable of handling big datasets. Although there is a limitation on the number of nodes in the original Hadoop implementations (and the largest reported Hadoop cluster includes about 4,000 nodes [71]) there are approaches to overcome this [72].

2. **Fault tolerance:** Hadoop replicates the data to several nodes so that in case of a node failure, data can be retrieved.

3. **Data locality:** Hadoop distributes the input data among the nodes. Using this approach, it can take advantage of the data locality and process data near to where they reside.

A distributed filesystem (DFS) allows us to store data on multiple nodes, and it maintains the metadata of the files. The HDFS is a Java-based DFS and the underlying filesystem of the Hadoop platform. It is responsible for partitioning the data into blocks and distribute them among nodes. The HDFS also generates a certain number of replicas of each block to make the system resilient against storage or node failures. It consists of a *NameNode* host, which takes care of filesystem metadata such as the location of the data blocks and status of the nodes, and multiple *DataNodes* hosts that store the blocks. Fig. 4.1 illustrates an overview of a HDFS.

On top of the HDFS, MapReduce [73] takes advantage of the partitioned data (i.e., data locality) to run map and reduce functions and orchestrate the cluster nodes to run distributed applications while data movements are minimalized. Fig. 4.2 shows a high-level overview of a MapReduce application on a Hadoop platform.

The MapReduce is a programming model that is able to process a large volume of data in a distributed fashion. The data should be formatted to key-value pairs to be processable by the

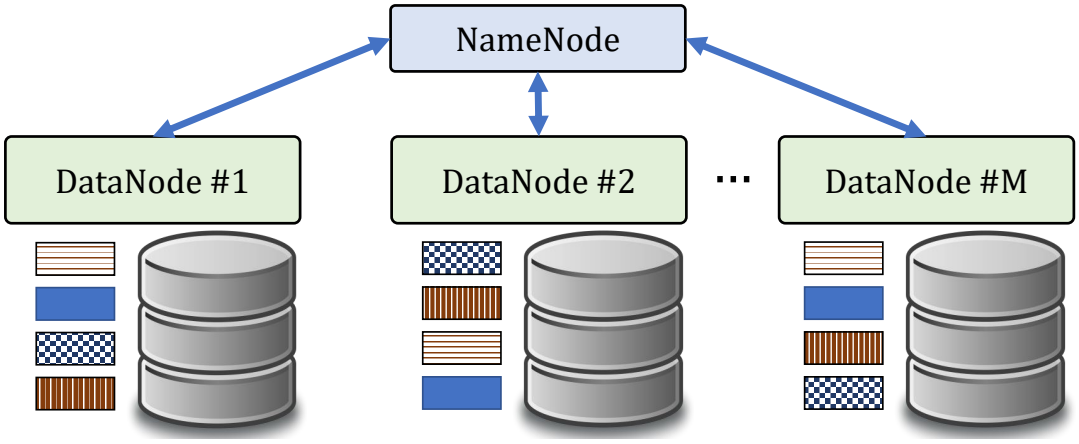


Figure 4.1: An overview of Hadoop filesystem (HDFS)

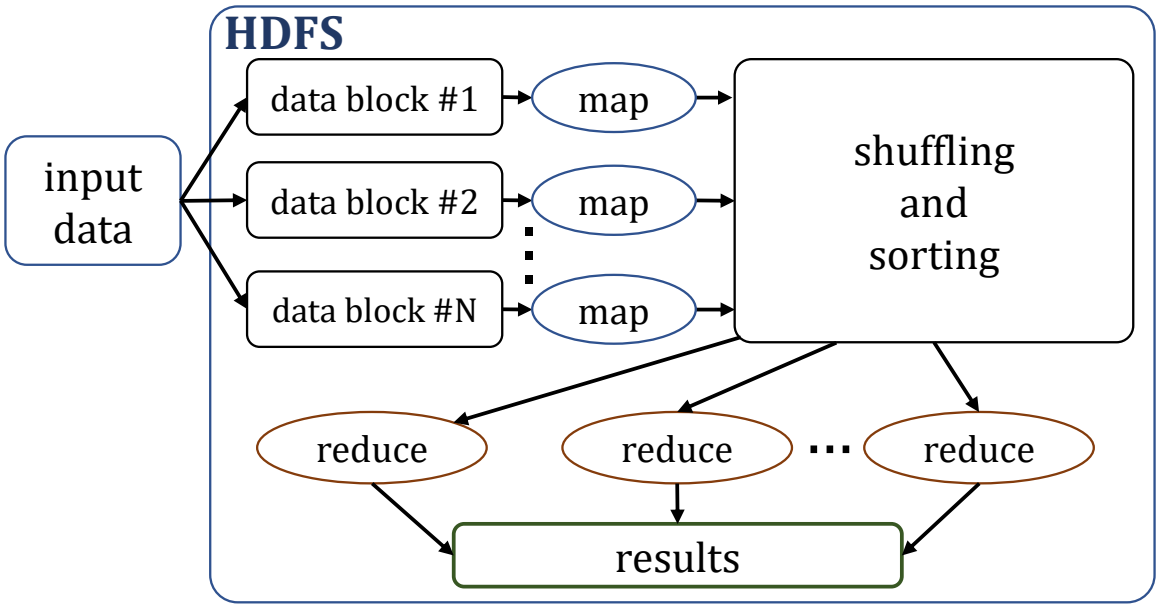


Figure 4.2: MapReduce application on Hadoop cluster

MapReduce-based distributed systems. This programming model has two functions (stages), namely, *map* and *reduce*. These two stages are performed sequentially in MapReduce. These two stages can be explained as follows:

- ***Map* stage:** This stage is composed of running Mapper processes on different nodes of the cluster. The Mappers initially bring the input data to the key-value format and then process the formatted data based on the *map* function. The Mappers process one key-value pair at a time. The output of the Mappers is also in the key-value format and can be smaller or larger than the input data. This output data is sent to the *reduce* stage.

- ***Reduce* stage:** The output of the Mappers should be sent to the Reducers; however, between these two main stages, the data should be shuffled and sorted. In fact, the output of the Mappers is sorted for each key, based on the value fields. Thus, the Reducers receive them in a sorted format. The Reducers consume the sorted data and generate the final output, which is saved in the HDFS.

According to Fig. 4.2, MapReduce is both a sequential and parallel programming model. In other words, although there are potentially a large number of Mappers and Reducers running in a parallel mode, Reducers can only start after the Mappers finish.

The Apache MapReduce 2 (Yarn) is one of the well-known MapReduce platforms [74]. Yarn agents manage the procedure of running Mappers, performing shuffle and sort, and running the Reducers to generate the output of the MapReduce application. The most important agents in the Yarn framework are a global resource manager (RM), one node manager (NM) per cluster node, and an application master (AM) per MapReduce application. The structure of the Yarn framework is shown in Fig. 4.3 (adapted from [75]).

The RM has a list of all the resources available in the cluster and manages the high-level

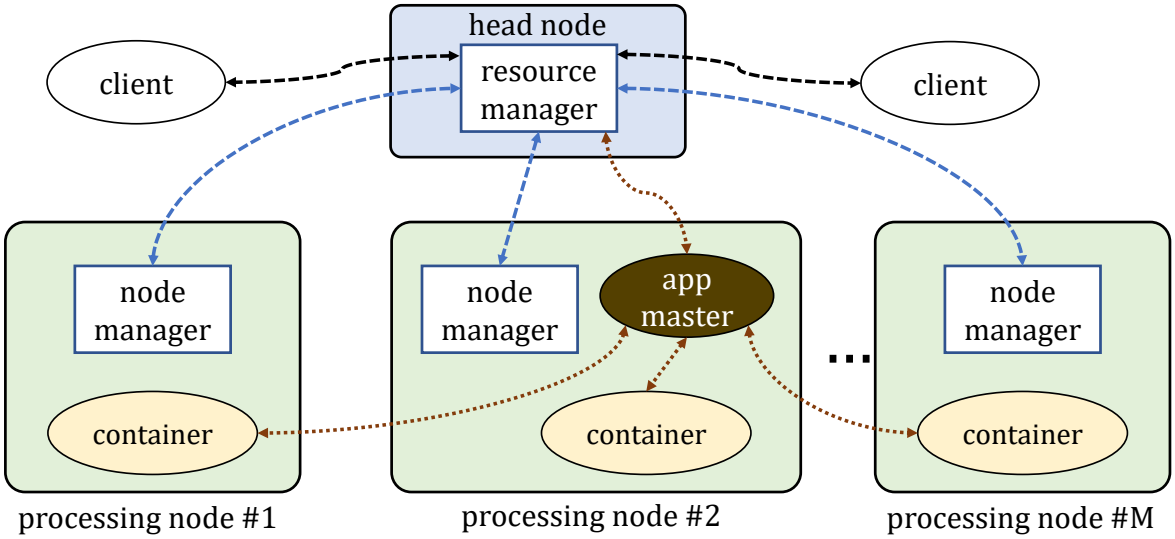


Figure 4.3: Apache Yarn structure

resource allocations to MapReduce applications. Meanwhile, *NM*s that run on the cluster nodes manage the local hosts' resources. The *RM* regularly talks to the *NM*s to manage the resources and poll the status of the nodes. Usually, the *RM* runs on the cluster *head node*, the same node that also runs the HDFS *NameNode*. However, each Hadoop cluster processing node runs a *NM* together with an HDFS *DataNodes*.

For each MapReduce application, an *AM* is created. The *AM* communicates with the *RM* to report the progress and status of the application. In a Yarn framework, a *container* is a virtual entity with limited resources that can run a Mapper or a Reducer. The user can define multiple containers on each of the processing nodes, based on the available resources of the nodes. For example, if a cluster processing node (in Fig. 4.3) has 8 GB of main memory and 4 cores dedicated to running MapReduce applications, the user can define 4 Hadoop *container*s on this cluster processing node, each with 2 GB of main memory and 1 core. These containers will be dynamically assigned to Mappers and Reducers of the applications. The *AM* is responsible for observing the containers that are assigned to the corresponding MapReduce application.

Overall, the Hadoop framework is composed of the HDFS and MapReduce platforms. To

process data in this framework, data should initially be imported to the HDFS. This initial process includes partitioning the input data into blocks, duplicating them, and storing them in the *DataNodes*. At this point, the data blocks are ready to be processed in a distributed fashion. Since Mappers preferably process the local data blocks, the MapReduce framework is known for bringing the process closer to the data in order to improve the energy efficiency and performance of the applications.

A MapReduce application targets a set of data blocks as well as the user-defined *map* and *reduce* functions. The procedure starts by running the Mappers on the targeted data blocks. The Mappers run concurrently on the Hadoop *DataNodes*, consume data blocks, and produce a set of key-value pairs to be used as the input of the Reducers. These intermediate key-value pairs are stored locally on the *DataNodes* and should be shuffled, sorted, and then transferred to the Reducers. The Hadoop framework stores the output of the Reducers in the HDFS, and, subsequently, it can later be imported to a host's local filesystem.

The Hadoop strategy of "processing data close to where they reside" is completely aligned with the ISP paradigm [8]. Thus, they can fortify each other's benefits when both are deployed concurrently in a cluster. In other words, Hadoop-enabled CSDs can play both roles of storage units for the conventional nodes as well as the ISP-enabled *DataNodes* simultaneously. This results in the augmentation of the processing horsepower of the CSDs to the Hadoop cluster.

Although CSDs can improve the overall performance of MapReduce applications by augmenting their processing engine to the Hadoop framework, this is not the primary advantage of deploying CSDs in the clusters. In other words, increasing the total horsepower of a cluster can also be achieved by adding more commodity nodes to a cluster. In fact, what makes CSDs distinguishable is the utilization of the high-performance, power-efficient internal data links of modern SSD architecture to run Hadoop MapReduce applications.

Moreover, well-designed CSDs can be deployed to run HPC applications in-place. However, CSDs need to deliver a compelling performance when running HPC applications; otherwise, it is hard to justify the complexity of deploying CSDs in the clusters while their performance improvement is not satisfactory. In this research, we argue that CSDs can considerably improve the performance of HPC applications when they utilize ASIC- or FPGA-based accelerators.

MPI for HPC applications

The term HPC refers to the use of powerful machines along with sophisticated parallel processing techniques to run heavy tasks in more efficient ways, both time-wise and energy-wise. HPC dates back to the 1960s, when high processing power became attractive for scientific projects [76]. The cost of a powerful machine was significantly higher than that of commodity ones. This led to the emergence of a new trend: using a cluster of commodity machines instead of a very powerful machine [77].

Although this technique is effective, it brought up new challenges such as communication bottlenecks between processing nodes and storage systems [78] and led to efforts to optimize interconnect systems for HPC applications [79]. Nowadays, HPC clusters utilize hundreds to millions of multicore CPUs and GPUs, providing trillions of floating-point operations per second (FLOPS) processing horsepower [80].

The MPI is a standardized parallel programming interface that allows multiple nodes in a cluster to run a task distributedly [81]. In fact, processes with separated address spaces can be connected using MPI for both synchronization and moving data from one host to another. MPI has four significant features as follows:

- **Standardized:** MPI is a well-known standard message passing library that is sup-

ported in almost all HPC platforms.

- **Portability:** There is no need to modify an MPI-based application to run on different platforms that are compatible with the MPI standard.
- **Completeness:** There are a large number of functions available in the MPI libraries. These functions provide a complete toolset for MPI programmers.
- **Availability:** There are many implementations of the MPI standard, such as Open-MPI [82] and MPICH [83].

The MPI with the features mentioned above is an acceptable programming tool when multiple CSDs and hosts run HPC applications. It also supports the heterogeneity that comes with the utilization of CSDs in the clusters. Thus, we used this parallel programming library to run different compute-intensive benchmarks on ISP-enabled systems.

4.1.2 Cluster Filesystems

The filesystem-level data access inside CSDs has a great advantage for developing applications as well as reusing the applications that are developed for conventional systems. However, providing filesystem-level access inside CSDs is very challenging, and to the best of our knowledge, there is no proposed CSD architecture that provides filesystem-level data access inside the ISP engine of CSDs. This challenge is rooted in two main issues: 1.) to support filesystem-level data access inside CSDs, there should be an operating system running in the storage unit. This requires a lot of time and resources to develop a CSD architecture that is able to run an OS inside. 2.) Although running an OS inside the CSD can provide filesystem-level access to the applications that run in-place, this will cause contention between the host's OS and the CSD's OS. In fact, the same flash memory will be mounted in two different operating systems concurrently.

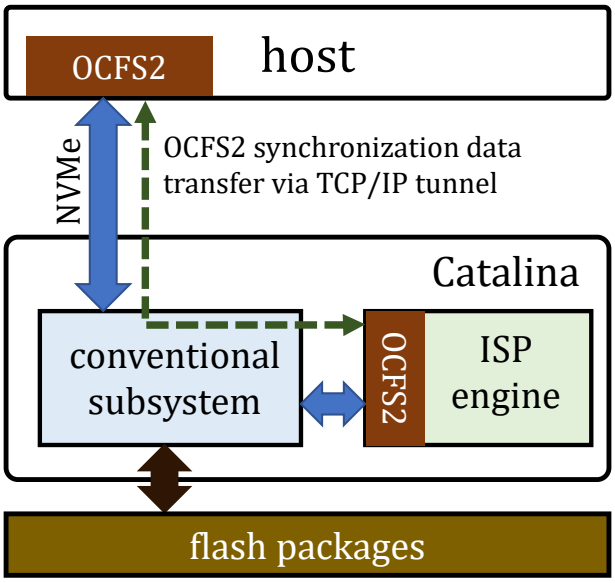


Figure 4.4: OCFS2 implementation in Catalina CSD

The concurrent access to the same flash memory from two operating systems can be addressed simply by assigning different partitions to the operating systems. In other words, their access to the flash memory will be limited to a portion of the flash memory. In this case, no data can be shared between the host and CSD simultaneously. However, the host can still write data in a partition and unmount it; then, the ISP engine inside the CSD can mount the partition and process the data in-place. This mechanism is time-consuming, especially when the host needs to modify the data repeatedly, which will cause many mount and unmount operations.

The cluster filesystems (CFS) can solve this problem by providing a synchronizing mechanism between the filesystem-level accesses of the operating systems. Using the CFS, both operating systems can mount the shared flash memory natively. This is the main difference between the CFS and network filesystem (NFS) [61]. Currently, many CFS are available, such as the IBM general parallel filesystem (GPFS) [84], Red Hat global filesystem (GFS) [85], Lustre [86], GlusterFS [87], and Oracle cluster filesystem 2 (OCFS2), which is a shared-disk cluster filesystem for high-performance and highly available systems [60].

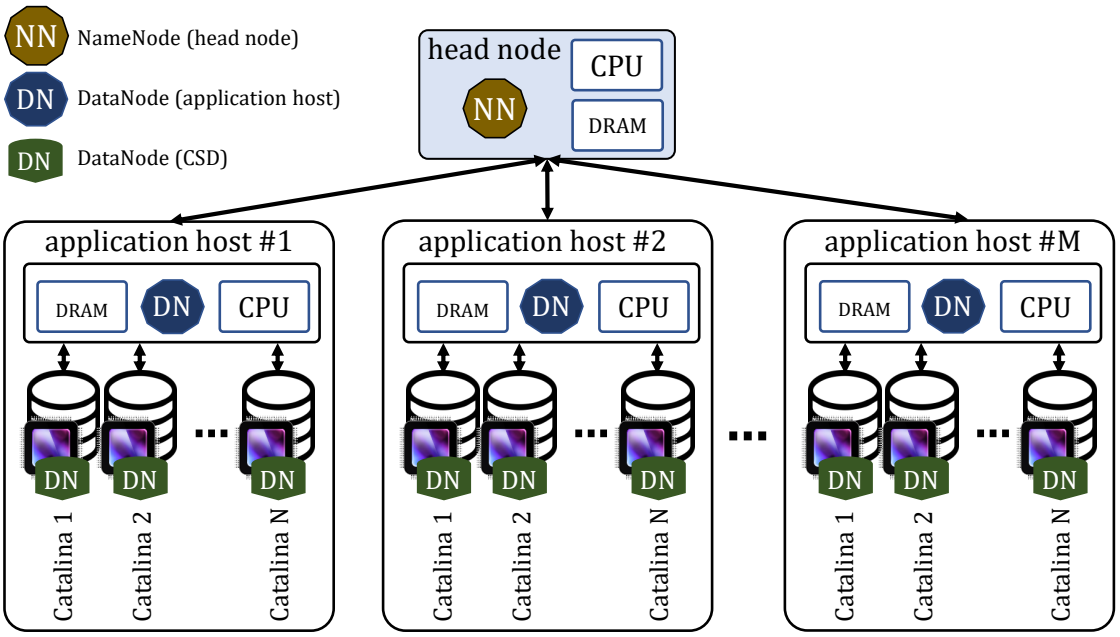


Figure 4.5: Overview of ISP-enabled Hadoop cluster

We chose the OCFS2 since it provided the functionality we needed to share the flash memory by both the host and ISP engine inside the Catalina CSD. Fig. 4.4 shows how the OCFS2 filesystem synchronization can be utilized in the Catalina CSD. It is noteworthy that the OCFS2 is, indeed, a cluster synchronization filesystem, which means it can be deployed on top of conventional filesystems such as Linux extended filesystems (i.e., ext2, ext3, and ext4). In other words, utilization of the OCFS2 cluster filesystem does not limit the user's choice of the underlying filesystem in the host.

4.2 Deploying CSDs in Distributed Platforms

Fig. 4.5 and Fig. 4.6 illustrate the ISP-enabled Hadoop and MPI-based clusters, respectively, where a *head node* is connected to M host machines and each of the hosts is equipped with N Catalina CSDs. In such a cluster, all CSDs and conventional nodes orchestrate together to improve the performance and efficiency of the distributed applications.

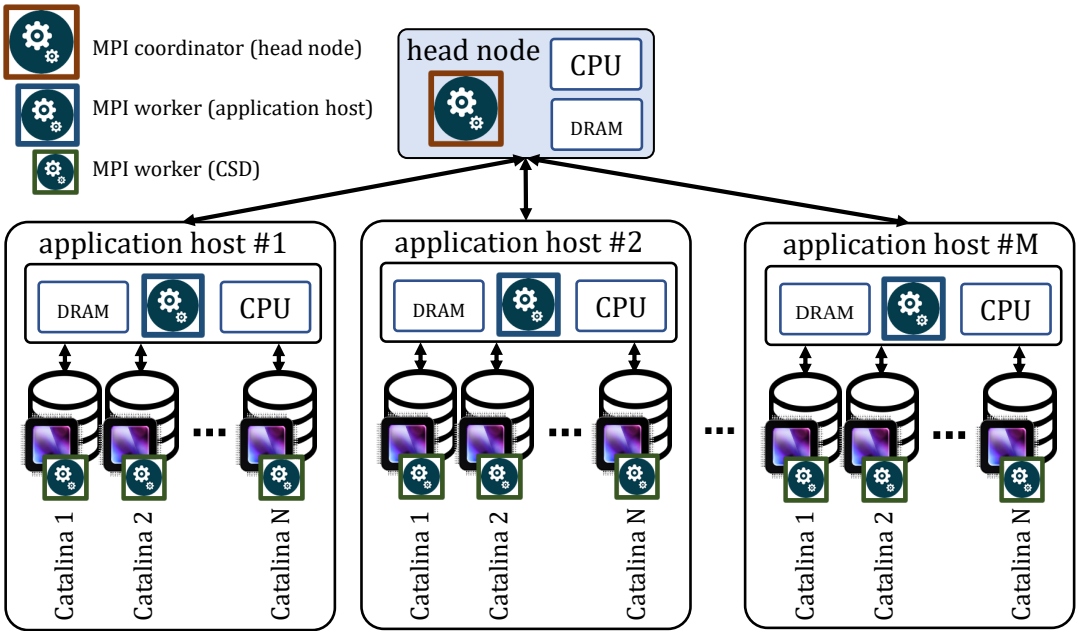


Figure 4.6: Overview of ISP-enabled MPI-based cluster

In the ISP-enabled Hadoop cluster, the *head node* runs the Hadoop *NameNode* and Yarn *RM*, while the hosts and Catalina CSDs run the *DataNodes* and *NM*s. In fact, the Catalina CSDs play the roles of both storage units and efficient *DataNodes*. Since Hadoop implements its filesystem synchronization mechanism, we did not need the OCFS2 filesystem to run Hadoop.

In Fig. 4.6, the *head node* runs an MPI coordinator, while the conventional hosts and the Catalina CSDs run the MPI workers. In this MPI-based cluster, each host is attached to N CSDs, and the data stored on the CSDs are shared between the host and CSDs so that the MPI workers on the host and CSDs have access to the shared data. Due to the OCFS2 filesystem, the shared data is simultaneously visible to the host and CSDs at the filesystem-level, so the user can freely distribute the processing loads among the hosts and CSDs.

In the remainder of this section, we will first demonstrate the developed platforms equipped with up to 16 Catalina CSDs and describe how we implemented an ISP-enabled Hadoop and MPI-based clusters on the developed platforms. Then, the second subsection will show the

performance and energy consumption results of running different Hadoop MapReduce and HPC benchmarks and discuss the benefits of deploying Catalina CSDs in clusters.

4.2.1 Experimental Setup

The Catalina CSDs were not designed to compete with the modern hosts that utilize high-end x86-based processors with tens to hundreds of gigabytes of DRAM. Instead, they were developed as a resource that augments the processing horsepower of a system and improves the performance and energy efficiency of the applications. To gain considerable improvements, we propose attaching multiple Catalina CSDs to host machines. Fig. 4.7 shows the architecture of the developed platform, which contains 16 Catalina CSD prototypes. We built this platform to investigate the benefits of deploying Catalina CSDs in clusters.

This platform is composed of a conventional host (the *head node*) and an *application host* which is equipped with the Catalina CSDs. These two hosts, along with the Catalina CSDs, form a distributed environment for running Hadoop MapReduce and MPI-based HPC applications. We use the *head node* exclusively for running Hadoop *NameNode* and the MPI coordinator to eliminate the load of the administrative tasks on the processing nodes. In other words, the *application host* and the CSDs are the processing nodes, while the *head node* is dedicated only to running the administrative tasks.

To extensively investigate the benefits of Catalina CSDs in different environments, we have considered three different configurations for the *application host*, namely *low*, *medium*, and *high*. The specifications for the *head node* and the different *application host*'s configurations are summarized in Table 4.1. In order to attach up to 16 Catalina CSDs to the *application host*, we used a Cubix Xpander Rackmount unit [88], which provides 16 PCIe Gen3 slots. This unit and the attached Catalina CSDs are shown in Fig. 4.7.

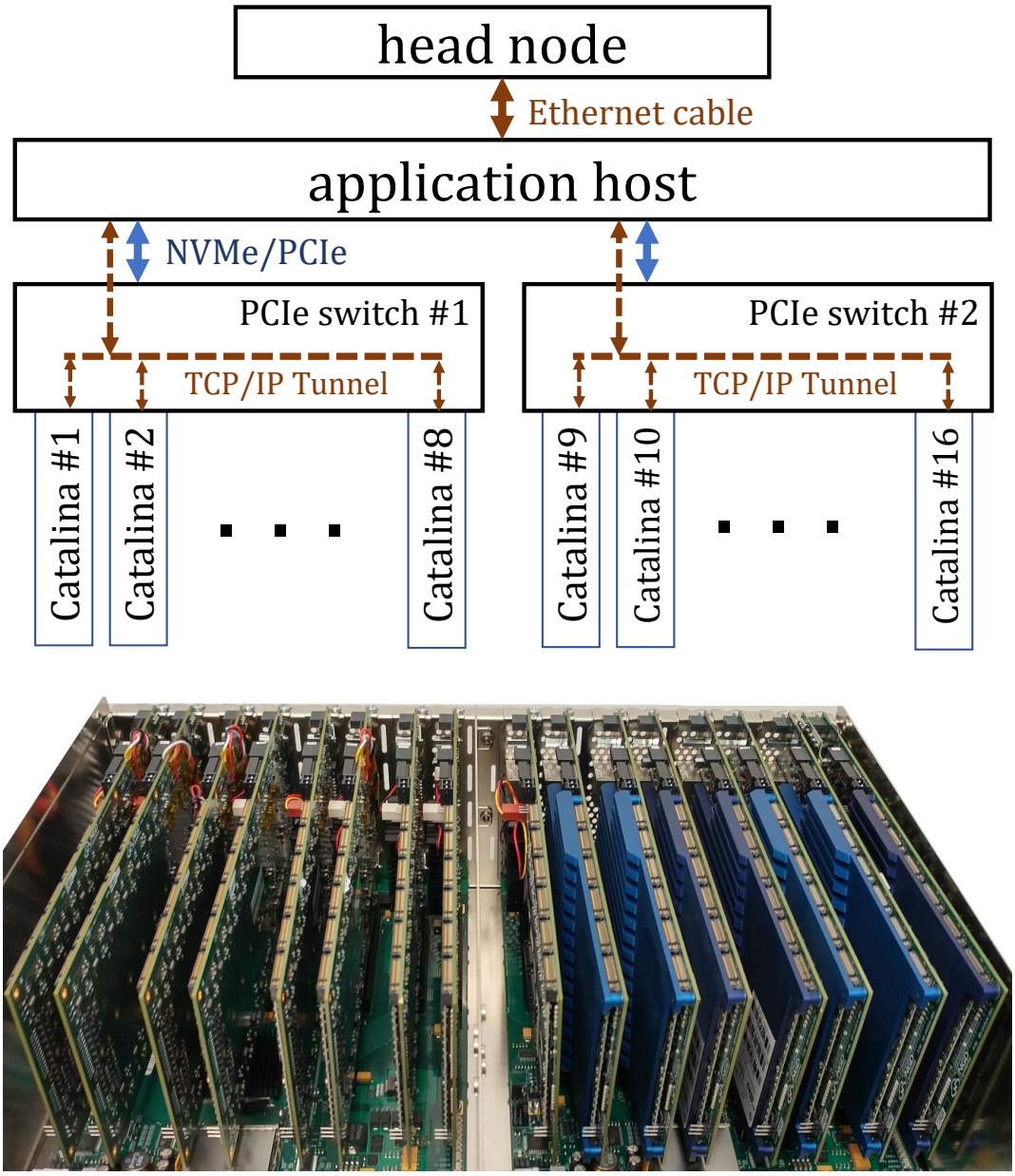


Figure 4.7: Architecture of the developed system equipped with 16 Catalina CSDs

Table 4.1: Specifications of the hosts in the developed system

feature	head node	application host configurations		
		low	*medium*	*high*
Processor	Xeon E5-2620 v4	Core i3-8100T	Core i7-7700	Xeon E5-2620 v4
Memory	32 GB (DDR 4)	32 GB (DDR 4)	32 GB (DDR 4)	32 GB (DDR 4)
Storage	4x Samsung 850 pro 1 TB SSD	6x Catalina CSD	6x Catalina CSD	16x Catalina CSD
CSD devices	None	6x Catalina CSD	6x Catalina CSD	16x Catalina CSD

The implementations of the Hadoop and MPI-based clusters are aligned with the architectures shown in Fig. 4.5 and Fig. 4.6, respectively. To implement the Apache Hadoop cluster, we ran the Hadoop *NameNode* and the Yarn *RM* on the *head node*, while the *DataNodes* and the YARN *NMs* were run on the *application host* and the Catalina CSDs attached to the *application host*. The communication between the *head node* and the *application host* was through an Ethernet cable, while the Catalina CSDs communicated via the developed TCP/IP over NVMe link.

However, to run the HPC application based on the MPI framework, we used the *head node* to run the MPI coordinator task, which initiated and organized the MPI worker tasks that run on the *application host* and the Catalina CSDs. In this case, the OCFS2 filesystem synchronized the filesystems of the Catalina CSDs and the *application host*, so at any given time the *application host* could access the entire data stored on all the CSDs directly, while each CSD only had access to its local data.

4.2.2 Benchmarks and Results

This section is composed of two subsections. First, we will describe the targeted Hadoop MapReduce benchmarks and report the performance and energy consumption of running the benchmarks for different configurations. Then, we will show the results of running 1D, 2D, and 3D discrete Fourier transform (DFT) algorithms utilizing the Neon SIMD engines of the Catalina CSDs. To report the performance, we measured the total execution time of running a benchmark on the developed platforms. To measure the energy consumption, we used a power meter to measure the power consumption of the platform. Using the logging tool provided by the power meter, we calculated the total energy consumption of running the benchmarks. However, we deducted the idle energy consumption from the total energy consumption for all the experiments to eliminate the energy consumption imposed by miscellaneous devices such as the cooling system.

Hadoop MapReduce benchmarks and results

To run Apache Hadoop MapReduce applications on the developed platform, we used a subset of the Intel HiBench benchmark suite [89] that includes Sort, Terasort, and Wordcount benchmarks. We believed that extensive experiments using these three benchmarks could show the potentials of the proposed CSD architecture for running Hadoop MapReduce applications. These benchmarks were executed on 16 different platform configurations, which are listed in Table 4.2. In all the experiments, the *head node* configuration was fixed and matches with Table 4.1, and the numbers of Mappers and Reducers tasks were 2000 and 200, respectively.

The *application host* used all the attached Catalina CSDs as the storage units (6 CSDs in the *low* and *medium* configurations, and 16 CSDs in the *high* configuration), while in each configuration, a certain number of the ISP engines of the CSDs were enabled to run the

Table 4.2: Different configurations for running Hadoop MapReduce benchmarks

experiment number	***application host* configuration**	**the enabled in-storage processing capability**
1	*low*	none
2	*low*	2 ISP-enabled CSDs
3	*low*	4 ISP-enabled CSDs
4	*low*	6 ISP-enabled CSDs
5	*medium*	none
6	*medium*	2 ISP-enabled CSDs
7	*medium*	4 ISP-enabled CSDs
8	*medium*	6 ISP-enabled CSDs
9	*high*	none
10	*high*	2 ISP-enabled CSDs
11	*high*	4 ISP-enabled CSDs
12	*high*	6 ISP-enabled CSDs
13	*high*	8 ISP-enabled CSDs
14	*high*	10 ISP-enabled CSDs
15	*high*	12 ISP-enabled CSDs
16	*high*	16 ISP-enabled CSDs

MapReduce application in-place. This way, the scalability of deploying the Catalina CSDs in clusters could be investigated. The data sizes for the Sort, Terasort, and Wordcount benchmarks were 8 GB, 1.3 GB, and 80 GB, respectively.

For the sake of accuracy, each experiment was repeated 30 times, and the performance and energy consumption results reported in this subsection are the average numbers of all repetitions. We ran the three targeted MapReduce benchmarks on the 16 different platform configurations, and each experiment was repeated 30 times, giving us a total of 1,440 MapReduce experiments.

As previously stated, in all experiments, the *application host* used all connected Catalina CSDs as storage units. However, in each test, a certain number of CSDs were enabled to run MapReduce application in-place and play the role of a processing node. Fig. 4.8 and Fig. 4.9 show the performance and energy consumption results, respectively, of the Hadoop MapReduce experiments.

The diagrams in Fig. 4.8 show that increasing the number of ISP-enabled CSDs decreased the elapsed time for all benchmarks. The performance of the *high*-configured *application host* platform increased up to 2.2x when the ISP engines of all 16 Catalina CSDs were enabled. Thus, deploying ISP-enabled CSDs increased the performance of the Hadoop MapReduce benchmarks significantly.

Moreover, according to these diagrams, the elapsed time for running the MapReduce benchmarks on the *low*-configured *application host* platform equipped with six Catalina CSDs was close to the elapsed time of running the benchmarks on the *high*-configured *application host* platform with no enabled ISP engine. Thus, only six Catalina CSDs could improve the performance of a low-end host close to the performance of a high-end host.

Also, Fig. 4.8 shows as we increased the number of ISP-enabled CSDs, the performance of the *low*-configured *application host* platform improved faster than the *medium*- and *high*-

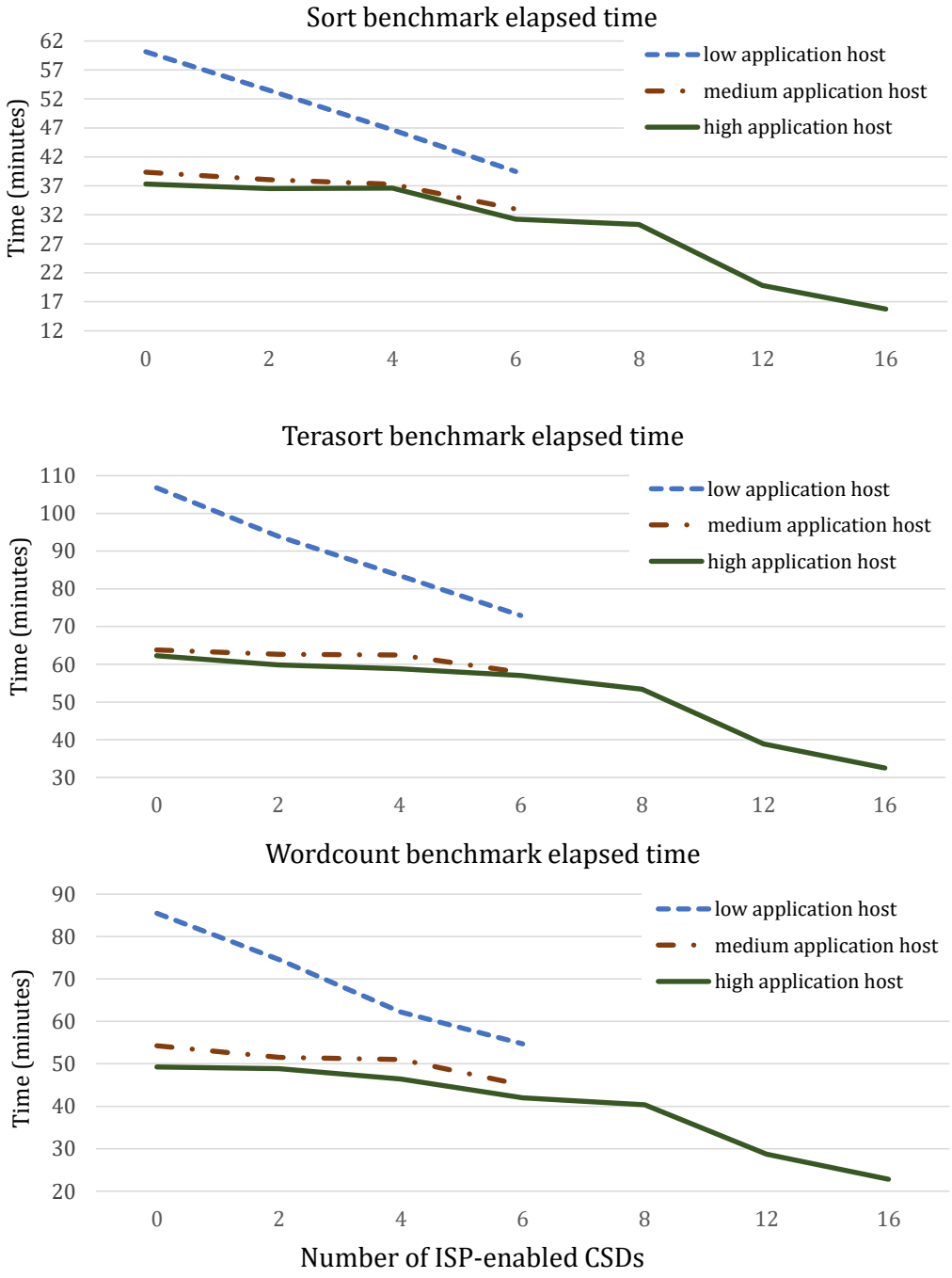


Figure 4.8: Hadoop MapReduce benchmarks performance results

configured *application host* platforms. In other words, ISP-enabled CSDs make a better improvement when their performance is significant in comparison to the host's CPU. If a host has a high-end CPU, augmenting CSDs only makes sense when the CSDs can deliver a compelling performance gain. In other words, if CSDs can run some applications in a high-performance mode by utilizing FPGA- or ASIC-based accelerators, they can be effectively augmented to more hosts with low- or high-end CPUs.

In Chapter 3, adding more CompStor CSDs led to a linear performance improvement for running compression and text search benchmarks. For those experiments, we distributed the data among all of the CSDs, and there was no communication between the CSDs. On the other hand, in the Hadoop platforms, the CSDs need to talk to each other when they run the *map* and *reduce* tasks, and this communication overhead may prevent the system to show a linear performance improvement when more CSDs are enabled to run applications in-place. Thus, as Fig. 4.8 shows, the performance improvement diagrams for the *medium*- and *high*-configured *application host* platforms are not linear for running Hadoop MapReduce benchmarks.

However, in Fig. 4.8, the performance diagram of the *low*-configured *application host* platform shows almost a linear behavior. We believe that the difference between the behaviors of the *low*-configured *application host* platform and other platforms for running the Hadoop MapReduce benchmarks is due to the relation between the CSDs' performance contribution and the communication overhead. In fact, the CSDs add processing horsepower to the whole system, but if they need to communicate, this causes an overhead which should be compensated by the augmented processing horsepower. In the *low*-configured *application host* platform, the CSDs improved the performance of the whole system more considerable than the other platforms, and the communication overhead was compensated properly.

As previously discussed, the cost of implementing the ISP engine inside the SSDs is negligible compared to the total cost of manufacturing an SSD, so ISP technology can considerably

improve the performance of Hadoop clusters economically. Fig. 4.9 shows the energy consumption results of running the Hadoop MapReduce benchmarks on the developed platform for different configurations.

According to Fig. 4.9, the energy consumption of running the benchmarks on the *low*-configured *application host* platform decreased up to 36% upon deploying 6 ISP-enabled Catalina CSDs. This improvement for the *high*-configured *application host* platform equipped with 16 ISP-enabled Catalina CSDs reached 4.3x.

With no ISP engine enabled, the *low*-configured *application host* platform was less energy efficient than the other configurations. This was expected behavior, since running the same benchmark on a more powerful platform takes less time. According to the diagrams in Fig. 4.9, when we enabled six ISP engines, the energy efficiency of the *low*-configured *application host* platform could surpass the energy efficiency of the *medium*- and *high*-configured *application host* platforms equipped with the same number of ISP-enabled CSDs. However, the performance of the *low*-configured *application host* platform was still lower than the other platforms (see Fig. 4.8). We believe that this occurred because of the high energy efficiency of the Catalina CSDs.

The Hadoop framework distributes tasks among all of the processing nodes. If a processing node gets idle, it will fetch data from other busy nodes and process it. Thus, the amount of data processed by each node in the Hadoop cluster is proportional to its processing resources. This means that in the *low*-configured *application host* platform, a larger amount of data is processed by the Catalina CSDs compared to the amount of data processed by them in the *high*-configured *application host* platform, which has an Intel Xeon processor. Since ISP engines are considerably more energy-efficient than the *application host*'s processor, as we increased the portion of data processed by the CSDs, the whole platform became more energy-efficient. This justifies why the energy efficiency of the *low*-configured *application host* platform equipped with six ISP-enabled CSDs could surpass the energy efficiency of the

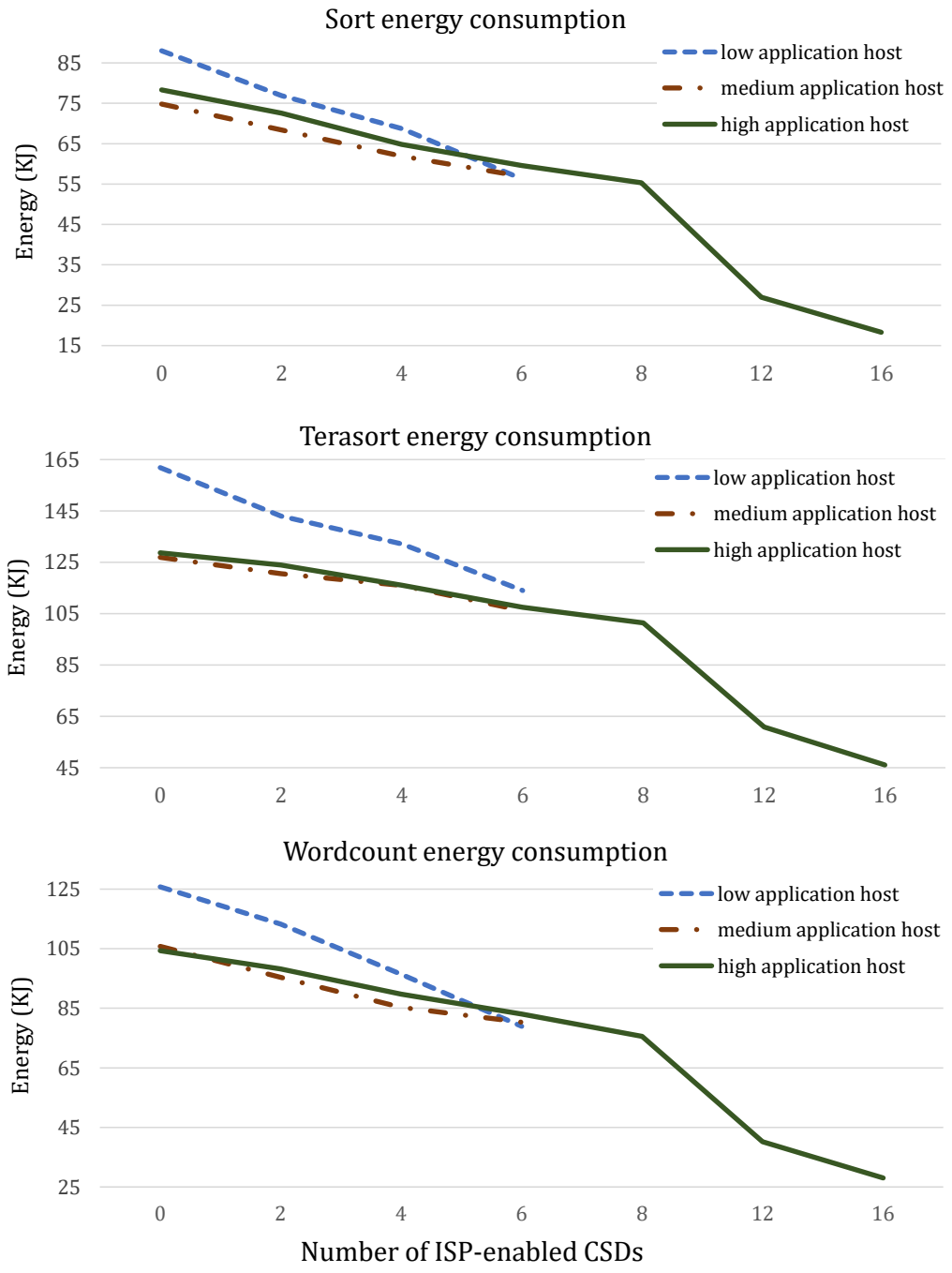


Figure 4.9: Hadoop MapReduce benchmarks energy consumption results

high-configured *application host* platform with the same number of ISP-enabled Catalina CSDs.

HPC benchmarks and results

In this section, we will first describe the targeted benchmarks to investigate the effect of deploying Catalina CSDs in clusters for running HPC applications. Then, we will show and discuss the performance and energy consumption results of running the benchmarks on the developed platform. HPC applications usually demand a considerable amount of processing resources and consume a large amount of data. Thus, we only considered the *high*-configured *application host* platform equipped with 16 Catalina CSDs to run the HPC experiments (see Table 4.1). We implemented the MPI framework to run the HPC benchmarks according to the architecture described earlier in this section. The MPI coordinator runs on the *head node* host, while the *application host* and the ISP-enabled Catalina CSDs run the MPI workers. In the developed platform, the *application host* can access the data stored on all the Catalina CSDs; however, each CSD only has access to its local data.

In addition, to run the HPC applications in-place, Catalina CSDs should be able to deliver a compelling performance. Therefore, we utilized the Neon SIMD engines inside the Catalina CSDs. The Neon SIMD engines are ASIC-based accelerators that are expected to improve the performance and energy efficiency of the applications significantly. Overall, this section will show how using ASIC-based accelerators enhances the benefits of deploying CSDs for running HPC applications [90].

The HPC Challenge benchmark suite [91], which was developed by the University of Tennessee, is a well-known HPC benchmark suite that has been used in many research works [92, 93, 94]. This suite is composed of several benchmarks, each of which focuses on a particular feature of the HPC clusters, such as the ability to do floating-point calculations, the

communication speed between nodes, and the potentials of running demanding algorithms such as DFT.

Among these benchmarks, we targeted the DFT algorithm since it is a CPU-intensive algorithm that also consumes a large amount of data, so it can show the potentials of deploying CSDs in clusters. Additionally, DFT is one of the most important algorithms, as Gilbert Strang, the author of the textbook *Linear Algebra and Its Applications* [95], referred to it as "the most important numerical algorithm in our lifetime." The DFT of a finite sequence X is a finite sequence Y with the same length of X in a complex-valued format in the frequency domain. The DFT of the finite sequence $X$is defined by (4.1).

$$\begin{aligned} Y &= F\{x_n\} \\ y_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi i}{N} kn} \end{aligned} \tag{4.1}$$

In the case of the multidimensional input signal of $X : \{x_{n_1,n_2,\cdots,n_l}\}$, a d-dimensional DFT is defined as (4.2).

$$\begin{aligned} y_{k_1,k_2,\cdots,k_1} &= \sum_{n_1=0}^{N_1-1} \left(\alpha_{N_1}^{n_1 k_1} \sum_{n_2=0}^{N_2-1} \left(\alpha_{N_2}^{n_2 k_2} \cdots \sum_{n_d=0}^{N_d-1} \left(\alpha_{N_d}^{n_d k_d} \cdot x_{n_1,n_2,\cdots,n_d} \right) \right) \right) \\ Where \quad \alpha_{N_l} &= exp\left(\frac{-2\pi}{N_l} \right) \end{aligned} \tag{4.2}$$

Considering a large amount of floating-point input data, the multi-dimensional DFT calculation is a challenging CPU-intensive application and can show the potentials of the Catalina CSDs for running HPC applications. Thus, we targeted this algorithm to measure the energy consumption and performance of 1D-, 2D-, and 3D-DFT calculations of large datasets run-

ning on the *high*-configured *application host* platform with different numbers of ISP-enabled Catalina CSDs. To implement the DFT algorithm, we utilized the *FFTW* library [96], which can be compiled to use the Neon SIMD engines of Catalina CSDs and also supports the multi-threading capability of the processing nodes in the developed platform.

To run the 1D-, 2D-, and 3D-DFT calculations, we prepared three different datasets. The *PTB Diagnostic ECG* dataset was used for the 1D-DFT calculation. The *PTB Diagnostic ECG* is a set of ECG signals collected from healthy volunteers and patients with different heart diseases by Professor Michael Oeff, M.D., at the Department of Cardiology of University Clinic Benjamin Franklin in Berlin, Germany [97, 98]. We duplicated this dataset to generate 200 million 1D objects, each of which is a sequence of 180 floating-point numbers.

Regularly, 2D-DFT operations are performed on images; therefore, we generated 14.4 million synthetic grayscale images for the 2D-DFT dataset. On each of these images, a dark point was placed randomly on the image, and other points' brightness was relative to their distance from the single darkest point. Fig. 4.10 shows four samples of these images. To perform the 2D-DFT operations, we converted each of the images to a 50×50 matrix. Overall, the 2D-DFT dataset was composed of 14.4 million 2D objects, each of which was a sequence of 2,500 floating-point numbers.

The 3D dataset was also generated using the same method we used to generate the 2D dataset. Each object in the 3D dataset can be described as a cube-shaped 3D object, wherein a single darkest point was placed randomly in the cube-shaped object, and other points' brightness is relative to their distance from the single darkest point. We generated a set of 288,000 three-dimensional objects and converted them to $50 \times 50 \times 50$ matrices to represent the dataset for the 3D-DFT operations. Table 4.3 summarizes the datasets we used to run the 1D-, 2D-, and 3D-DFT operations on the developed ISP-enabled platform.

Similar to the Hadoop MapReduce experiments, in all of the DFT calculation experiments,

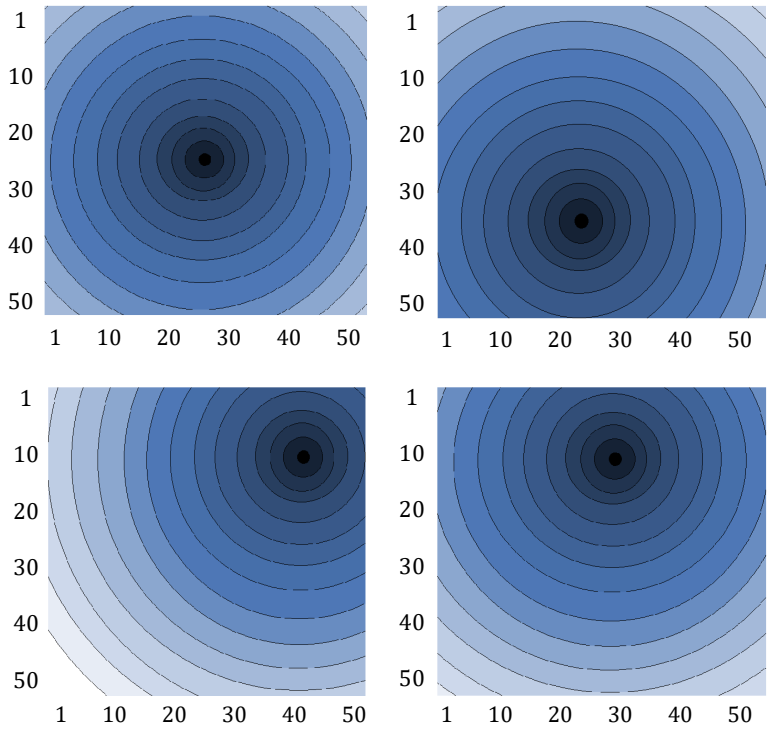


Figure 4.10: Four images of the 2D-DFT dataset

Table 4.3: Datasets for 1D-, 2D-, and 3D-DFT calculations

dataset	number of objects	dimensions of an object	total size of the dataset
1D-DFT	200 million	180 × 1	288 GB
2D-DFT	14.4 millions	50 × 50	288 GB
3D-DFT	288,000	50 × 50 × 50	288 GB

the *application host* had access to the data stored in all of the Catalina CSDs, and the CSDs always played the role of storage units. However, in each test, a certain number of ISP engines of the CSDs were enabled to show the scalability of ISP technology for running HPC applications. Fig. 4.11 shows the performance and energy consumption results of running the DFT calculations on the developed platform for different numbers of ISP-enabled Catalina CSDs. The performance reported in the diagrams is defined as the number of 1D, 2D, and 3D objects that were processed in a second, and the reported energy consumption is the energy consumed for processing an object. It is worth mentioning that each test was repeated 20 times, and each result reported in this subsection is the average of all repetitions.

According to the diagrams in Fig. 4.11, as we enabled more ISP engines, the performance increased, and the energy consumption decreased. In these experiments, adding 16 ISP-enabled Catalina CSDs improved the performance and energy consumption of running DFT calculations by factors of 5.4x and 8.9x, respectively. The comparison between the results of running the Hadoop MapReduce and HPC benchmarks yielded an important outcome. The deployment of Catalina CSD in the platform improved the performance and energy consumption of running DFT calculations significantly more than the Hadoop MapReduce benchmarks. We believe that this difference is rooted in the utilization of the Neon SIMD engines in running the DFT calculations. In other words, the Neon SIMD engines accelerated the execution of the DFT algorithms considerably. Since in Catalina CSDs, these engines are close to where the data reside, they made a compelling improvement when the ISP engines utilized them for running the applications in-place.

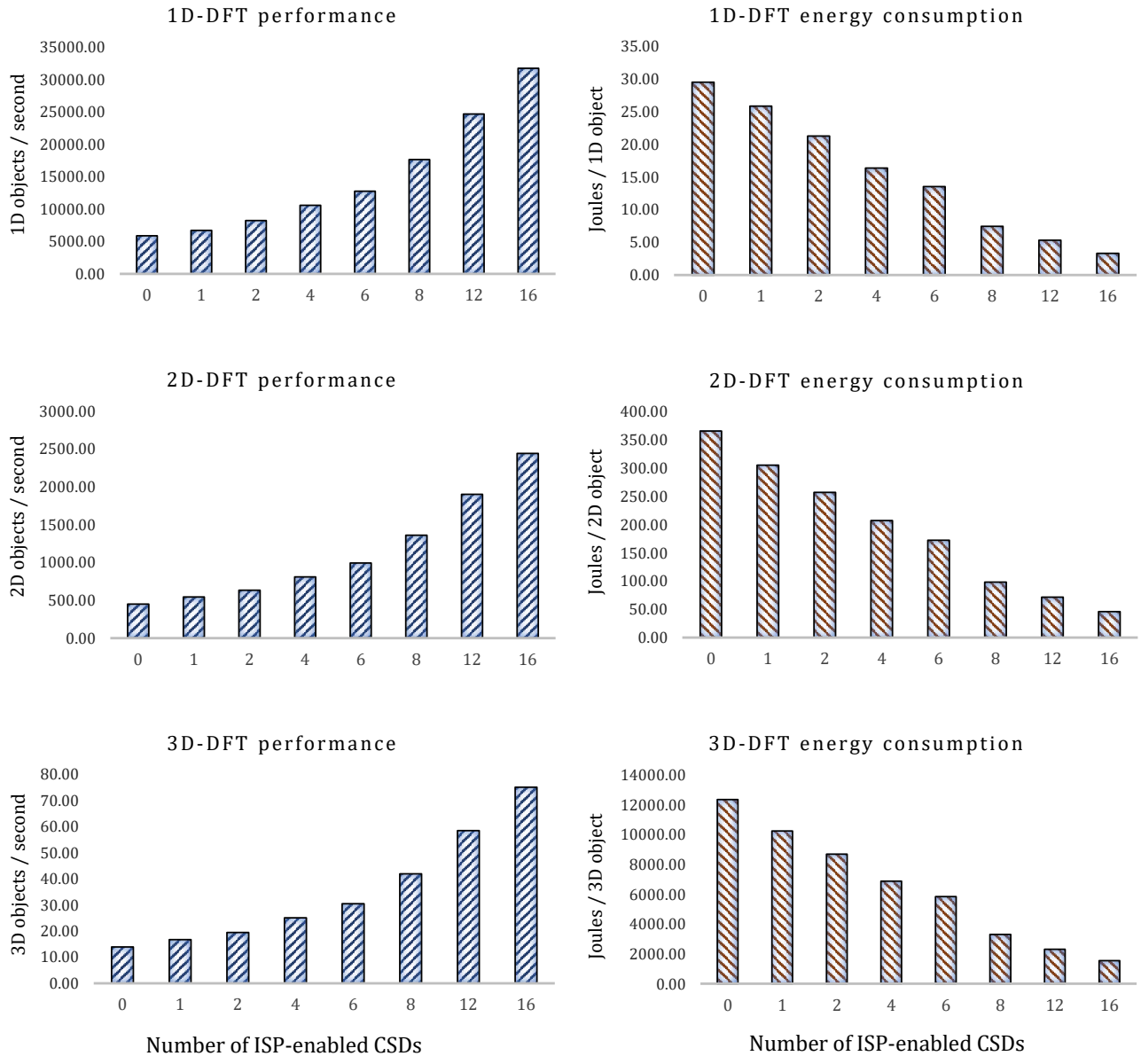


Figure 4.11: DFT experiments performance and energy consumption results

Chapter 5

FPGA-Based Acceleration for ISP

One of the most significant features of Catalina is the capability to provide FPGA-based accelerators inside the ISP engine. Some papers reported significant performance improvements by implementing some parts of applications in the FPGAs [99]. Additionally, since all the ISP components, including the accelerators, are implemented inside the same chip, such an accelerator can improve the performance and energy efficiency of applications considerably. In this chapter, we will implement an FPGA-based accelerator inside Catalina and investigate the benefits of the CSDs that are equipped with FPGA-based accelerators for running highly demanding applications.

This chapter is composed of three sections, as follows: The first section describes the architecture of the FPGA-based accelerator that we developed for enhancing the performance and energy efficiency of the targeted application. The second section defines the application we chose to run on the developed system, and, finally, the experimental results that show the effectiveness of the proposed solution appear in the third section. In fact, this chapter aims to show the potential of implementing an FPGA-based accelerator inside Catalina CSDs.

5.1 An FPGA-Based Accelerator Inside Catalina

Matrix multiplication is a basic operation in many applications, including physics, economics, statistics, and machine learning applications [100]. This operation is highly demanding when the input matrices are oversized, and the elements are floating-point numbers. Equation 5.1 shows a simple matrix multiplication of a 4×4 matrix A and a 4×2 matrix B, which results in a 4×2 matrix C.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \\ c_{41} & c_{42} \end{pmatrix} \tag{5.1}$$

$$A \times B = C$$

In Equation 5.1, c_{11} equals $(a_{11}.b_{11}) + (a_{12}.b_{21}) + (a_{13}.b_{31}) + (a_{14}.b_{41})$, which is composed of four floating-point multiplication and three floating point addition operations. In other words, to calculate one column of matrix C, 28 floating-point operations should be done (16 multiplications and 12 additions).

As a result, the total number of floating-point operations for calculating the multiplication of a 4×4 matrix and a $4 \times N$ matrix is equivalent to $28 \times N$. We developed an FPGA-based accelerator in Catalina that is able to do this matrix multiplication in only $2 \times N$ clock cycles. The accelerator's working frequency is 250 MHz, which means it can perform $125 million \times 28$ floating-point operations per second. Thus, the performance of the FPGA-based accelerator is about 3.5 GFlops, and it consumes data at a rate of up to 2 GBps. To reach such a high performance, we utilized the AXI stream protocol for data transfer inside the accelerator's architecture.

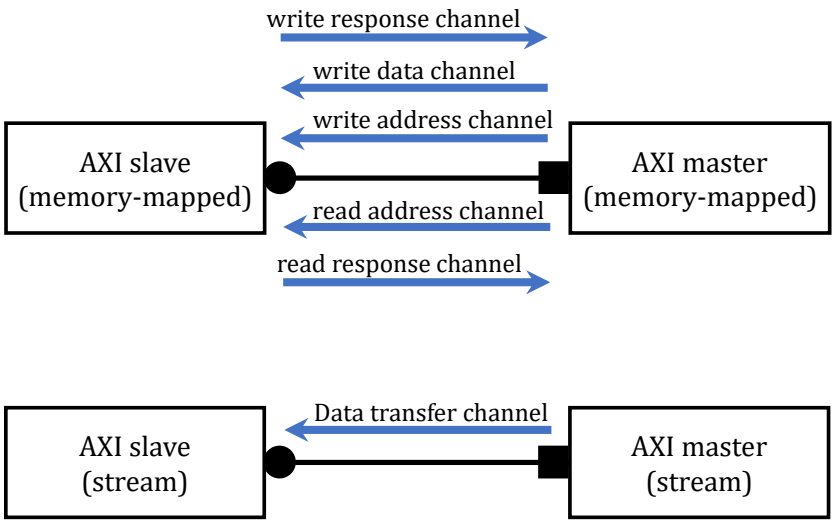


Figure 5.1: AXI memory-mapped versus AXI stream data transfer channels

The AXI is a high-performance master-slave ARM advanced microcontroller bus architecture [59]. There are two different AXI protocols available in Xilinx Zynq Ultrascale plus chip [101], namely AXI memory-mapped and AXI stream. The AXI memory-mapped is suitable when there are multiple modules sharing a bus and data should have source and destination addresses to be routed from one module to another. However, the AXI stream is designed for high-speed data transfer between two modules when the data do not have to be addressed, e.g., a stream of video frames. Fig. 5.1 shows the data channels of the AXI memory-mapped and AXI stream protocols. The AXI stream is lighter, so data can potentially be transferred faster.

The developed FPGA-based accelerator multiplies a 4×4 single-precision floating-point matrix to a $4 \times N$ matrix in a pipelined design, where N can be any integer. The main module of the accelerator is a vector multiplier that calculates the inner product of two four-dimensional (4D) floating-point vectors. The architecture of this module, which is called a *floating-point vector multiplier*, is shown in Fig. 5.2.

The module depicted in Fig. 5.2 implements the multiplication of two 4D vectors. The building blocks of the *floating-point 4D vector multiplier* module are a "streaming floating-

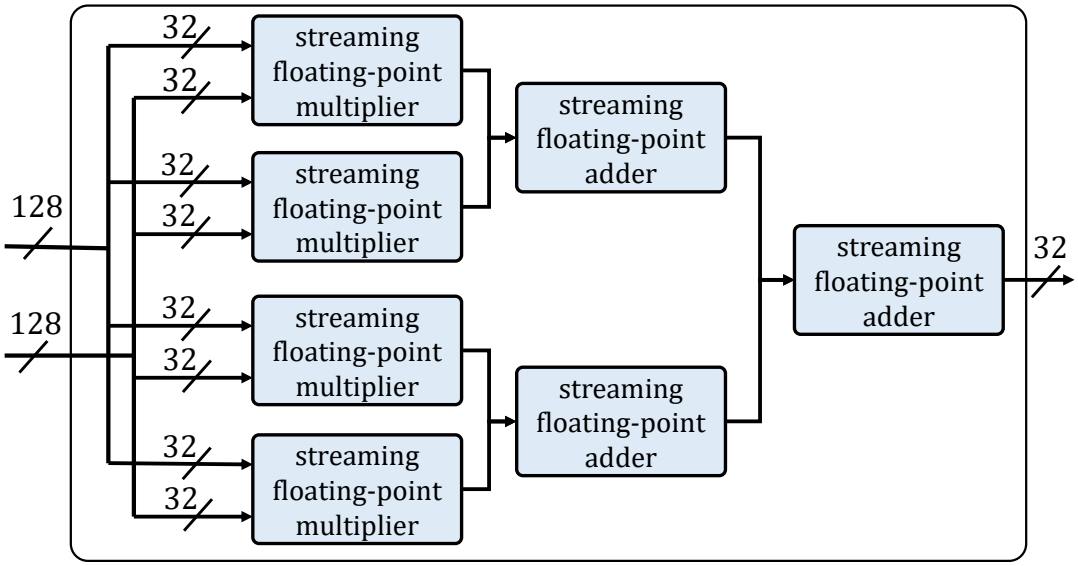


Figure 5.2: Architecture of a *floating-point 4D vector multiplier* block

point adder" and a "streaming floating-point multiplier." These two blocks are Xilinx soft intellectual properties that are implemented in a heavily pipelined mode and can consume data through AXI stream interfaces. The module shown in Fig. 5.2 is replicated four times to generate the main FPGA-based accelerator. The overall architecture of the FPGA-based accelerator is illustrated in Fig. 5.3.

Since we have utilized a Xilinx Zynq Ultrascale plus MPSoC chip, we need to make the connection between the quad-core ARM Cortex-A53 processor and the FPGA-based accelerator through the high-performance AXI ports available between the PS and PL subsystems of the chip (see Fig. 3.11).

Since all the components in the PS subsystem, including the DRAM memory and ISP engine, are in a memory-mapped space, we first need to convert the memory-mapped data to stream data. Thus, a direct memory access unit (DMA) transfers data from the DRAM to a *memory map to stream convertor* module. As mentioned before, the floating-point multipliers are AXI stream interfaced; thus, it is possible to feed the accelerator at a very high data rate. Finally, the stream output data is converted to the AXI memory-mapped format, and the

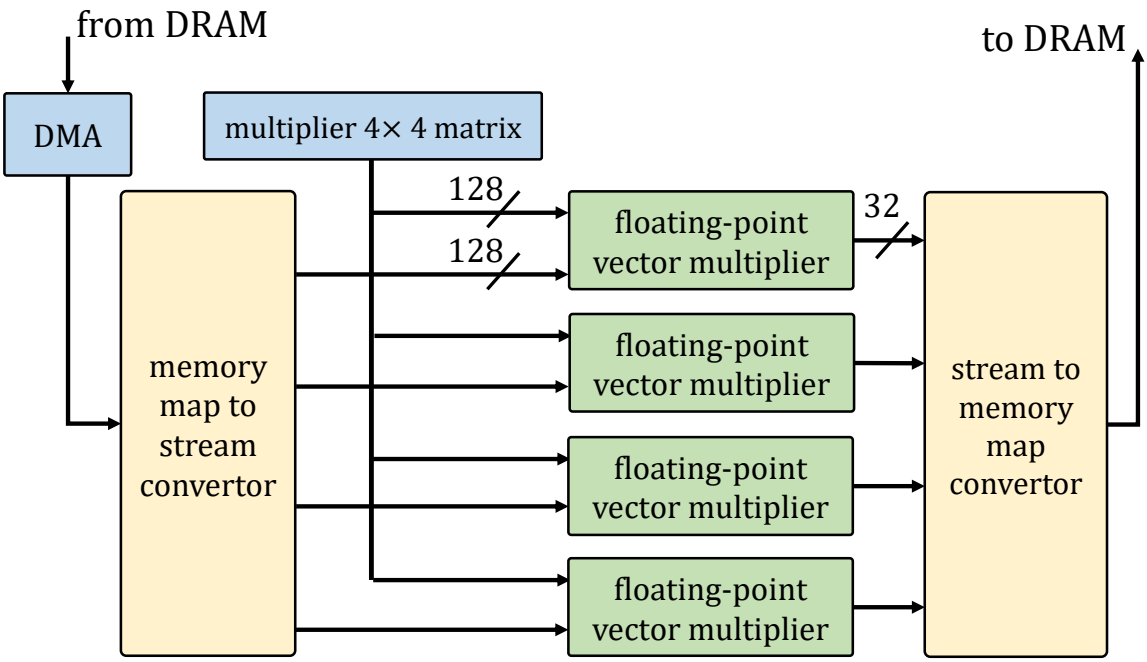


Figure 5.3: Architecture of the FPGA-based accelerator

result is directly written back to the DRAM. The developed FPGA-based accelerator makes it possible to read the input matrices at a high rate from the DRAM memory and write back the results to the memory. This accelerator consumes 7% of the configurable logic blocks (CLBs), 3.5% of lookup tables (LUTs), and less than 3% of the DSP blocks of the Zynq Ultrascale plus MPSoC chip.

5.2 Running Image Similarity Search In-Place

For the evaluation of the proposed solution, we used an open-source library called Faiss, which was originally developed by Facebook AI Research [102, 103]. Faiss is a library for image similarity search, i.e., it allows users to search for multimedia documents that are similar to each other in a dataset. To utilize Faiss in an image similarity search application, each multimedia item (e.g., image, sound, or video) should be represented by a vector. Two vectors are similar to each other when they are close in the Euclidean space. The Euclidean

distance between two items is calculated by an inner product of the two vectors that represent the items. Since the vectors are defined in a high-dimensional space and each dimension is defined as a floating-point number, each member of the dataset is usually a tuple composed of at least 128 floating-point numbers (128D). To compute the distances between these high-dimensional vectors, the inner products between them should be calculated, which is a compute-intensive task, especially when these computations have to be made in very large datasets.

Faiss includes different algorithms for the purpose of similarity searches like *flat*, *IVF*, and *product quantization* (PQ) [104]. The *flat* is a brute-force algorithm that searches the entire dataset to find exact matches. The other algorithms search only a portion of the dataset, which makes them faster but less accurate compared to the *flat* algorithm. However, regardless of the algorithm, the computationally intensive part of all of them is the inner product of a large number of vectors. The developed device driver for the FPGA-based accelerator provides an API set to do the inner product of very large floating-point vectors.

5.3 Experimental Results

In this section, we evaluate the benefits of using FPGA-based accelerators to improve the energy-efficiency and performance of ISP-enabled storage systems. In the following subsection, the experimental setup will be described, and we will discuss the architecture of the developed platform. Then, the energy consumption and performance of the proposed solution will be discussed.

Table 5.1: Dataset used in the similarity search application

test dataset	ANN_SIFT1B
dimension	128 floating-point numbers
search space set size	1 billion images
query set size	10 thousand images

5.3.1 Experimental Setup

To run the experiments in this chapter, we used the platform shown in Fig. 4.7 with the *high*-configured *application host* (see Table 4.1). We added the developed FPGA-based accelerator to the Catalina CSDs. In order to implement the image similarity search, we used the Faiss library to execute the compute-intensive task of finding similar images to a set of query images in a dataset with 1 billion images. Overall, the benchmark application searched for the 100 most similar items in the entire dataset for each of the query images. Such a search required a large number of the inner products of the two 128D floating-point vectors. In this experiment, we used the ANN-SIFT1B dataset [105]. The brute-force algorithm (*flat*) was chosen to be used in all the experiments. We did not explore other algorithms that trade search accuracy for performance because it did not fall within the scope of this research. For each experiment, two datasets were used: search space and query sets. Table 5.1 shows the details of these datasets.

The ANN-SIFT1B search space set is very large, and we distributed it among the *application host* and the Catalina CSDs. We used MPI as the distributed processing framework to process the data on the Catalina CSDs and the *application host*, which are the processing nodes (see Fig. 4.6). Since the whole search space set was distributed among the processing nodes, the queries were dispatched to all of them, and they ran the image similarity search in a parallel fashion as the MPI workers. Upon completion, the local results were sent to the *head node*, and then the MPI coordinator took care of aggregating the results.

5.3.2 Results

This subsection compares the performance and energy consumption of the benchmark application for seven different platform configurations. In all of the experiences, the *application host* used all 16 Catalina CSDs as the storage units, but the numbers of enabled ISP engines of Catalina CSDs, as well as the FPGA-based accelerators, were different. The platform configurations that we will compare in this subsection are as follows:

1. The *application host* with no ISP-enabled Catalina CSDs (Catalina CSDs used only as the storage units).
2. The *application host* with six ISP-enabled Catalina CSDs (software only – using the ARM Cortex-A53 processors). In this case, FPGA-based accelerators were not enabled.
3. Similar to the second configuration, but the FPGA-based accelerators of the ISP-enabled Catalina CSDs were enabled.
4. Eight ISP-enabled Catalina CSDs were used, but no FPGA-based accelerator was enabled.
5. Eight ISP-enabled Catalina SSDs with enabled FPGA-based accelerators.
6. Sixteen ISP-enabled Catalina CSDs were used, but no FPGA-based accelerator was enabled.
7. Sixteen ISP-enabled Catalina SSDs with enabled FPGA-based accelerators.

Fig. 5.4 shows that as we increased the number of ISP-enabled Catalina CSDs, the performance of the application increased. We achieved an 11x improvement in the performance of the image similarity search when all 16 Catalina CSDs were enabled to run the application in-place. This figure also shows the performance gain when the FPGA-based accelerators were

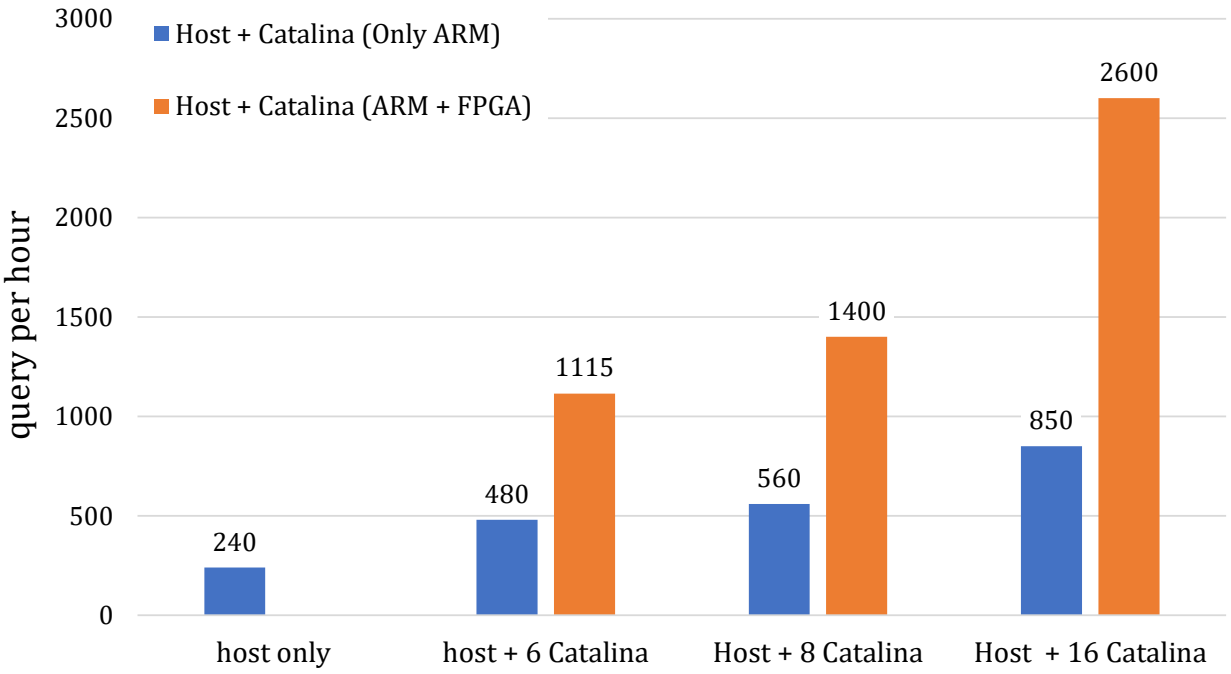


Figure 5.4: Performance results of the similarity search application

enabled in the CSD-equipped storage system. When the Catalina CSDs used FPGA-based accelerators, there was a 3x gain over the same experiment run on a similar configuration without using the FPGA-based accelerators.

The energy consumption was measured over the processing of all 10,000 image queries fed in at a rate that saturated the system. The energy consumption per query was then derived as considering an average over the total duration of the experiment. The experiment was performed in a similar way for all seven configurations mentioned above. Fig. 5.5 shows how the heterogeneous ISP approach (utilizing both the quad-core ARM Cortex-A53 processors and the FPGA-based accelerators) resulted in a 7x reduction in energy consumption per query. The energy consumption per query reduced from 825 J when the application was run exclusively on the *application host's* CPU down to 119.13 J when the *application host's* CPU was combined with the 16 ISP-enabled Catalina CSDs equipped with FPGA accelerators.

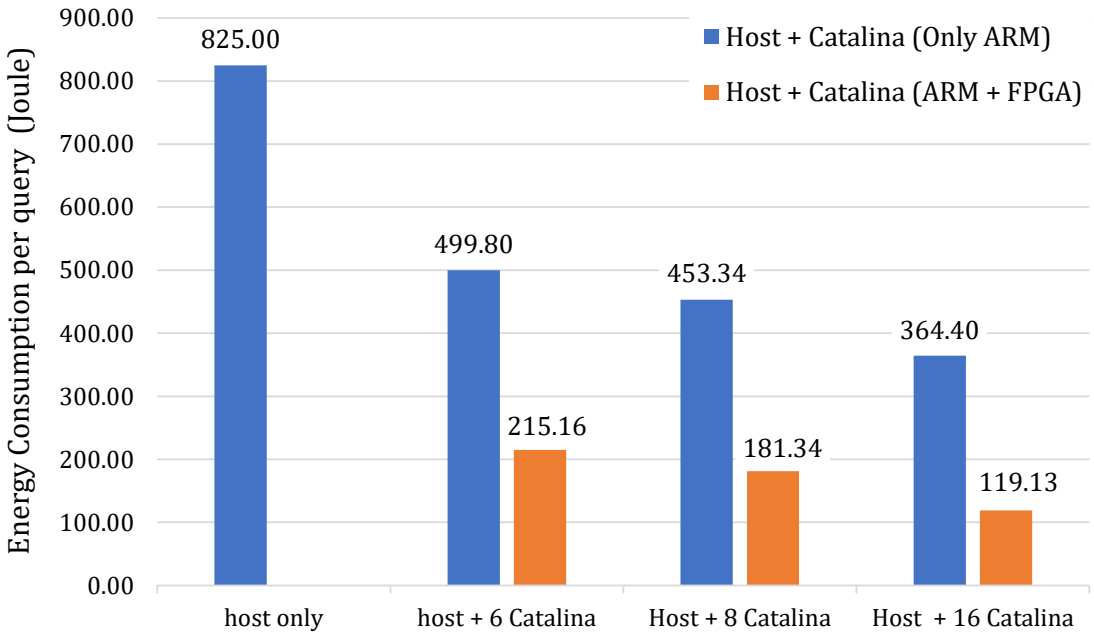


Figure 5.5: Energy consumption results of the similarity search application

Chapter 6

Conclusion and Future Works

In this chapter, we will first summarize our achievements throughout this research. Then, the future research topics will be discussed. These topics could potentially extend the contributions of this dissertation.

6.1 Summary

ISP technology enables storage units to run user applications in-place, i.e., data are not required to move from the storage units to the host's main memory to be processed. It can relieve the data movement challenges in highly demanding super-scale applications wherein huge data need to be fetched from the storage systems. The modern SSD architecture uses flash packages to store data, and they are faster and more efficient than the traditional HDDs that use magnetic disks. Regularly, a modern SSD controller includes multiple cores to run conventional flash management and host interface routines. Such an architecture provides a better environment for implementing ISP technology compared to the HDDs. The SSD architecture that is enabled to run user applications in-place and is equipped with

an ISP engine is called a CSD. This dissertation proposed efficient CSD architectures and investigated the benefits of deploying such CSDs for running different types of applications. We introduced two NVMe CSD architectures, namely CompStor and Catalina. These two CSD architectures both have a dedicated ISP engine that runs a full-fledged Linux operating system to provide a flexible environment for running user applications in-place.

CompStor is the first proposed CSD architecture that is composed of a conventional flash management subsystem and an ISP engine. These two subsystems are implemented on two boards that are connected via an FMC connector. We ran several compute- and I/O-intensive algorithms to explore the benefits of deploying CompStor CSDs in systems. The experimental results showed up to 2.6x and 3.3x improvements in performance and energy consumption, respectively, when running the applications on CompStor in comparison to the host's CPU.

In CompStor, the conventional subsystem controls the flash memory and accesses data stored in the flash memory chips. The data should move from the conventional subsystem to the ISP engine, which is implemented on two boards. Although this off-chip data link is less costly than a complex host interface link such as NVMe, data still need to move from one board to another to be processed, and this data transfer is not aligned with the core concepts of ISP.

To solve this problem, we introduced Catalina, which is equipped with a controller that includes both the conventional subsystem and the ISP engine implemented in an SoC. Thus, the ISP engine has access to the data stored in flash memory via a high-performance, power-efficient intra-chip data link. We developed a block device driver that abstracts the underlying ISP engine and the conventional subsystem data transfer, and applications running inside Catalina have filesystem-level access to the data stored in flash memory.

Additionally, a TCP/IP tunnel through NVMe over PCIe link was developed to allow

Catalina CSD to communicate with the host. Catalina can be seamlessly deployed in distributed environments such as Hadoop and MPI-based clusters. For the proof of concept, we built a fully functional Catalina CSD prototype as well as a system equipped with 16 Catalina CSDs to investigate the benefits of deploying the CSDs in clusters. The experimental results showed that the deployment of Catalina CSDs in the clusters improved the Hadoop MapReduce application's performance and energy efficiency up to 2.2x and 4.3x, respectively. By utilizing the Neon SIMD engines to accelerate DFT algorithms running in-place, the performance and energy efficiency improvements grew even further to 5.4x and 8.9x. Also, using FPGA-based accelerators, Catalina CSDs can improve the performance and energy consumption of a highly demanding image similarity search application up to 11x and 7x, respectively.

6.2 Future Directions

CSDs are more energy-efficient in comparison to the host's CPU for the following reasons. First, CSDs decrease the data movement between the host and the storage units dramatically. This data movement is usually through a complex data link such as NVMe over PCIe. Second, due to the limited power budget of the storage units, the ISP engines usually utilize a power-efficient embedded processor. These reasons make CSDs low-power environments for processing user data. However, CSDs also need to provide a compelling performance for different applications to justify the complexity that comes with deploying them in systems.

Since embedding very high-end processors inside CSDs is practically impossible due to the power and economical budgets of the storage units, the reasonable technique is to provide a heterogeneous environment inside CSDs to process data. Such an environment could benefit from FPGA- and ASIC-based accelerators to improve the performance of some applications that run in-place. This dissertation followed this method to propose efficient CSD architec-

tures. For future works, there could be multiple topics based on the heterogeneous SoC-based ISP method advocated for in this dissertation. Future directions are suggested as follows:

1. In both architectures that are proposed in this dissertation, the conventional flash management subsystem and the ISP engine are two subsystems that communicate with each other. The development of such architecture is less error-prone in comparison with a uniform design wherein both subsystems are integrated into a single system. For future works, a uniform design could be proposed that could potentially improve the performance and energy consumption of the CSD architectures.

2. The ASIC-based accelerators in Catalina are limited to the engines that are available in Xilinx Zynq Ultrascale plus MPSoC. This limits the applications that can benefit from this type of accelerator. Developing a CSD architecture with more ASIC-based accelerators could improve the performance of some applications that were not explored in this dissertation.

3. The possibility of implementing FPGA-based accelerators inside CSDs gives a considerable amount of flexibility to the CSD architecture. This flexibility comes with the time-consuming process of designing, synthesizing, and implementing the targeted accelerator inside the CSD. For example, the implementation of the matrix multiplier FPGA-based engine, which is discussed in Chapter 5, took a long time to reach a stable and error-free design. High-level synthesis (HLS) [106] tools could potentially shorten the required time for the design and implementation of the FPGA-based accelerators inside the ISP engines considerably. However, due to the presence of other components in a CSD architecture, adding an FPGA-based accelerator using HLS tools needs some considerations. One direction of the future works could concern the integration of HLS tools into the CSD design flow.

Bibliography

[1] The Economist. The world's most valuable resource is no longer oil, but data. *The Economist: New York, NY, USA*, 2017.

[2] Flávio Kapczinski, Benson Mwangi, and Ives Cavalcante Passos. *Personalized Psychiatry: Big Data Analytics in Mental Health*. Springer, 2019.

[3] Rob Kitchin and Gavin McArdle. What makes big data, big data? exploring the ontological characteristics of 26 datasets. *Big Data & Society*, 3(1):2053951716631130, 2016.

[4] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[5] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE micro*, 15(1):29–36, 1995.

[6] Prabhat and Quincey Koziol. *High Performance Parallel I/O*. CRC Press, Cleveland, Ohio, USA, 2014.

[7] George Eason, Benjamin Noble, and Ian Naismith Sneddon. On certain integrals of lipschitz-hankel type involving products of bessel functions. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 247(935):529–551, 1955.

[8] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., Sebastopol, California, USA, 2012.

[9] ARM, cortex-r series processors web page. https://developer.arm.com/ip-products/processors/cortex-r, Accessed on 20 December 2018.

[10] SATA ecosystem web page. https://sata-io.org, Accessed on 5 June 2019.

[11] Serial-attached scsi (SAS) web page. https://searchstorage.techtarget.com/definition/serial-attached-SCSI, Accessed on 5 June 2019.

[12] Non-volatile memory express project web page. https://nvmexpress.org, Accessed on 5 June 2019.

[13] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. Near-memory computing: Past, present, and future. *Microprocessors and Microsystems*, 2019.

[14] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11):81–91, 1998.

[15] Hyeran Lim, Vikram Kapoor, Chirag Wighe, and David H-C Du. Active disk file system: A distributed, scalable file system. In *2001 Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, pages 101–101. IEEE, 2001.

[16] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[17] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. A case for intelligent disks (idisks). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[18] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of active disks for decision support databases. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*, pages 337–348. IEEE, 2000.

[19] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 153–165. IEEE Press, 2016.

[20] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pages 1–12. IEEE, 2013.

[21] J Paul Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[22] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Information Sciences*, 327:183–200, 2016.

[23] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.

[24] Sang-Woo Jun, Ming Liu, Kermin Elliott Fleming, et al. Scalable multi-access flash store for big data analytics. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 55–64. ACM, 2014.

[25] Sungjin Lee, Jihong Kim, and Arvind Mithal. Refactored design of i/o architecture for flash storage. *IEEE Computer Architecture Letters*, 14(1):70–74, 2014.

[26] Tobias Vincon, Sergey Hardock, Christian Riegger, Andreas Koch, and Ilia Petrov. nativendp: Processing big data analytics on native storage nodes. 2019.

[27] The R project for statistical computing. https://www.r-project.org, Accessed on 27 April 2019.

[28] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[29] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. Noftl-kv: Tacklingwrite-amplification on kv-stores with native storage management. In *EDBT*, pages 457–460, 2018.

[30] Xiaojia Song, Tao Xie, and Wen Pan. Risp: a reconfigurable in-storage processing framework with energy-awareness. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 193–202. IEEE, 2018.

[31] Siavash Rezaei, Kanghee Kim, and Eli Bozorgzadeh. Scalable multi-queue data transfer scheme for fpga-based multi-accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 374–380. IEEE, 2018.

[32] Benjamin Y Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. Xsd: Accelerating mapreduce by harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing*, 2013.

[33] Roman Kaplan, Leonid Yavits, and Ran Ginosar. Prins: Processing-in-storage acceleration of machine learning. *IEEE Transactions on Nanotechnology*, 17(5):889–896, 2018.

[34] Seonyeong Park, Youngjae Kim, Bhuvan Urgaonkar, Joonwon Lee, and Euiseong Seo. A comprehensive study of energy efficiency and performance of flash-based SSD. *Journal of Systems Architecture*, 57(4):354–365, 2011.

[35] Jim Cooke. Micron technology, flash memory: An introduction to NAND flash. https://www.eetimes.com, Accessed on 2 August 2019.

[36] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, volume 57, 2008.

[37] The open compute project (OCP) web page. https://www.opencompute.org, Accessed on 18 November 2018.

[38] Samsung 960pro SSD specifications. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/960pro, Accessed on 28 July 2019.

[39] ADATA , XPG sx8200 pro 2280 SSD data sheet. https://www.adata.com/upload/downloadfile/Datasheet_XPG, Accessed on 26 July 2019.

[40] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 309–316, 2019.

[41] David Mayhew and Venkata Krishnan. Pci express and advanced switching: evolutionary path to building next generation interconnects. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 21–29. IEEE, 2003.

[42] Christoph Lameter et al. Numa (non-uniform memory access): An overview. *Acm queue*, 11(7):40, 2013.

[43] Rohit Gupta. Western digital , what is nvme and why is it important? a technical guide. https://blog.westerndigital.com/nvme-important-data-driven-businesses, Accessed on 24 April 2019.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[45] Seagate, kinetic hdd produce web page. https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd, Accessed on 20 September 2018.

[46] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. Compstor: An in-storage computation platform for scalable distributed processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1260–1267. IEEE, 2018.

[47] Xilinx , microblaze processor reference guide. https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0020-microblaze-hub.html, Accessed on 10 June 2019.

[48] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[49] ARM , cortex-a53 processor documentation web page. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53/docs, Accessed on 7 June 2019.

[50] Dave Barker. Vita technologies, introducing the fpga mezzanine card: Emerging vita 57 (fmc) standard brings modularity to fpga designs. http://vita.mil-embedded.com/articles/introducing-fpga-brings-modularity-fpga-designs, Accessed on 15 June 2019.

[51] ARM , cortex-r5 processor production web page. https://www.arm.com/products/silicon-ip-cpu/cortex-r/cortex-r5, Accessed on 2 June 2019.

[52] Shibamouli Lahiri. Complexity of word collocation networks: A preliminary structural analysis. *arXiv preprint arXiv:1310.5111*, 2013.

[53] L Peter Deutsch. Gzip file format specification version 4.3. https://www.gzip.org, Accessed on 2 September 2018.

[54] J Seward. A program and library for data compression. bzip2 and libbzip2. http://www.bzip.org, Accessed on 2 September 2018.

[55] Gnu grep project web page. https://www.gnu.org/software/grep, Accessed on 2 June 2018.

[56] Gnu awk project web page. http://www.gnu.org/software/gawk, Accessed on 2 June 2018.

[57] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader 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)*, pages 430–437. IEEE, 2019.

[58] Xilinx, zynq ultrascale+ mpsoc product web page. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html, Accessed on 25 August 2018.

[59] Christina Toole. ARM , introduction to AXI protocol: Understanding the AXI interface. https://community.arm.com/developer/ip-products/system/b/soc-design-blog/posts/introduction-to-axi-protocol-understanding-the-axi-interface, Accessed on 29 June 2019.

[60] Oracle, oracle cluster filesystem second version web page. https://oss.oracle.com/projects/ocfs2, Accessed on 5 February 2019.

[61] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, 2000.

[62] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent SSDs. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102. ACM, 2013.

[63] Trendforce , dramexchange web page. https://www.dramexchange.com, Accessed on 19 June 2019.

[64] Jose Luis Bosque and Luis Pastor. A parallel computational model for heterogeneous clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1390–1400, 2006.

[65] Aditya B Patel, Manashvi Birla, and Ushma Nair. Addressing big data problem using hadoop and map reduce. In *2012 Nirma University International Conference on Engineering (NUiCONE)*, pages 1–5. IEEE, 2012.

[66] Lena Mashayekhy, Mahyar Movahed Nejad, Daniel Grosu, Quan Zhang, and Weisong Shi. Energy-aware scheduling of mapreduce jobs for big data applications. *IEEE transactions on Parallel and distributed systems*, 26(10):2720–2733, 2014.

[67] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[68] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[69] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. 2003.

[70] C Long. Data science and big data analytics: Discovering, analyzing, visualizing and presenting data. *Indianapolis, Indiana*, 2015.

[71] Konstantin V Shvachko and Arun C Murthy. Scaling hadoop to 4000 nodes at yahoo. *Yahoo! Developer Network Blog,*, 2008.

[72] Ke Wang, Ning Liu, Iman Sadooghi, Xi Yang, Xiaobing Zhou, Tonglin Li, Michael Lang, Xian-He Sun, and Ioan Raicu. Overcoming hadoop scaling limitations through distributed task execution. In *2015 IEEE International Conference on Cluster Computing*, pages 236–245. IEEE, 2015.

[73] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[74] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[75] Apache, yarn project web page. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, Accessed on 10 September 2018.

[76] Eric Gottfrid Swedin and David L Ferro. *Computers: the life story of a technology*. Greenwood Publishing Group, 2005.

[77] Al Geist and Daniel A Reed. A survey of high-performance computing scaling challenges. *The International Journal of High Performance Computing Applications*, 31(1):104–113, 2017.

[78] Baruch Awerbuch, Rainer Gawlick, Tom Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 412–423. IEEE, 1994.

[79] Reza Asadi, Solmaz S Kia, and Amelia Regan. Cycle basis distributed admm solution for optimal network flow problem over biconnected graphs. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 717–724. IEEE, 2016.

[80] Hans Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. The top 500 list of supercomputers. https://www.top500.org, Accessed on 3 April 2018.

[81] David W Walker and Jack J Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[82] Open MPI: Open source high performance computing. https://www.open-mpi.org, Accessed on 9 December 2018.

[83] MPICH: High-performance portable MPI. https://www.mpich.org, Accessed on 9 December 2018.

[84] Jason Barkes, Marcelo R Barrios, Francis Cougard, Paul G Crumley, Didac Marin, Hari Reddy, and Theeraphong Thitayanun. GPFS: a parallel file system. *IBM International Technical Support Organization*, 1998.

[85] Steven R Soltis, Thomas M Ruwart, and Matthew T OKeefe. The global file system. 1996.

[86] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[87] Gluster filesystem web page. https://www.gluster.org, Accessed on 17 July 2019.

[88] Cubix, the xpander produce web page. https://www.cubix.com/xpander, Accessed on 2 June 2019.

[89] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[90] Mahdi Torabzadehkashi, Ali Heydarigorji, Siavash Rezaei, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. Accelerating HPC applications using computational storage devices. In *The 21st IEEE Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2019.

[91] The HPC challenge benchmark suite web page. http://www.hpcchallenge.org, Accessed on 10 March 2019.

[92] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[93] Jack Dongarra and Piotr Luszczek. Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling. *University of Tennessee Computer Science Technical Report, Tech. Rep*, 2010.

[94] Peter Steinbach and Matthias Werner. gearshifft–the FFT benchmark suite for heterogeneous platforms. In *International Supercomputing Conference*, pages 199–216. Springer, 2017.

[95] Nicholas J Rose. Linear algebra and its applications (gilbert strang). *SIAM Review*, 24(4):499–501, 1982.

[96] Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[97] R Bousseljot, D Kreiseler, and A Schnabel. Nutzung der ekg-signaldatenbank cardiodat der ptb über das internet. *Biomedizinische Technik/Biomedical Engineering*, 40(s1):317–318, 1995.

[98] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000.

[99] S. Rezaei, C. Hernandez-Calderon, S. Mirzamohammadi, E. Bozorgzadeh, A. Veidenbaum, A. Nicolau, and M. J. Prather. Data-rate-aware fpga-based acceleration framework for streaming applications. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Nov 2016.

[100] Henry Cohn, Robert Kleinberg, Balazs Szegedy, and Christopher Umans. Group-theoretic algorithms for matrix multiplication. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 379–388. IEEE, 2005.

[101] Xilinx , AMBA AXI4 interface protocol web page. https://www.xilinx.com/products/intellectual-property/axi.html, Accessed on 2 June 2019.

[102] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 2019.

[103] Facebook AI research, a library for efficient similarity search and clustering of dense vectors. https://github.com/facebookresearch/faiss, Accessed on 20 November 2018.

[104] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[105] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[106] Michael C McFarland, Alice C Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.