UNIVERSITY OF CALIFORNIA,
IRVINE

Leveraging Computational Storage for Power-Efficient Distributed Data Analytics

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Ali HeydariGorji

2024

DEDICATION

To my beloved family,
Mom, Dad, my sister, and my brother-in-law,
for your endless love, unwavering support, and constant encouragement throughout this journey.
This achievement is as much yours as it is mine.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

As I complete this PhD journey, I find myself reflecting on the many individuals and organizations who have supported and guided me along the way. Without their contributions, this work would not have been possible. I am deeply grateful to everyone who helped me during this journey.

First and foremost, I extend my heartfelt thanks to Professor Pai Chou, my PhD advisor and true friend. His invaluable guidance, unwavering support, and constant encouragement have been the cornerstone of my academic and personal growth. Professor Chou's expertise and advice were instrumental in overcoming the numerous challenges encountered during my research. His mentorship has made this achievement possible, and I am forever grateful for his friendship and dedication.

I would also like to express my sincere gratitude to Dr. Ruey-Kang Chang, MD, for his support of my initial research on embedded and wearable devices. His insights and encouragement provided a solid foundation for my work and set me on the path to success.

A special thank you goes to Dr. Vladimir Alves from NGD Systems for providing me with the opportunity to work on their cutting-edge devices. His support and the resources he made available were crucial in advancing my research. Collaborating with NGD Systems has been a remarkable experience, and I am grateful for the opportunity to contribute to their innovative projects.

I am deeply indebted to Mehdi Torabzadeh, Siavash Rezaei, and Hossein Bobarshad for their valuable contributions to my research. Their collaboration, insights, and technical expertise were essential in overcoming various challenges and advancing my work. Their support and friendship have been greatly appreciated throughout this journey.

I would also like to thank my thesis committee members, Professor Fadi Kurdahi and Professor Philip Sheu, for their invaluable feedback and guidance. Their expertise and constructive critiques have greatly enriched my work and helped me refine my research.

I extend my gratitude to the National Science Foundation (U.S.) for their support through NGD Systems under grant number 1660071. Their funding was instrumental in enabling the research and development of advanced technologies that were central to my work. Additionally, I am thankful to the National Institute of Health for their support through NeoVative, Inc. under grant 1R41GM113463-01A1. This support significantly contributed to the progress and success of my research.

To my Mom, Dad, Sister, and brother-in-law, thank you for your unwavering support and encouragement. Your belief in me has been a source of strength and motivation, especially during the most challenging times. Your love and sacrifices have made this accomplishment possible, and I am eternally grateful.

I am also thankful to my friends, who were always there for me. Your companionship, understanding, and encouragement have been a source of joy and comfort. Whether celebrating successes or providing a listening ear during difficult times, your support has meant the world to me.

To each and every one of you, thank you for making this journey possible. Your contributions, both big and small, have been integral to my success. I am deeply grateful for your support and encouragement.

VITA

Ali HeydariGorji

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering	**2024**
University of California, Irvine	*Irvine, California*
Master of Science in Electrical and Computer Engineering	**2018**
University of California, Irvine	*Irvine, California*
Bachelor of Science in Electrical Engineering	**2016**
Sharif University of Technology	*Tehran, Iran*

EXPERIENCE

Software Engineer	**2022–now**
Google	*Irvine, California*
Software Engineer	**2018–2022**
NGD Systems Inc.	*Lake Forest, California*
Graduate Research Assistant	**2016–2022**
University of California, Irvine	*Irvine, California*
System Designer Engineer	**2017–2017**
QT Medical	*Diamond Bar, California*
Circuit and PCB Designer	**2017–2017**
Cognitive Anteater Robotics Laboratory (CARL)	*Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant	**2017–2022**
University of California, Irvine	*Irvine, California*

Introduction to Digital Systems
Introduction to Electrical Engineering
Introduction to Digital Logic Laboratory
Processor Hardware/Software Interfaces

Teaching Assistant	**2015–2016**
Sharif University of Technology	*Tehran, Iran*

Logic Circuits & Digital Systems (+Lab)
Microprocessor System Design

REFEREED JOURNAL PUBLICATIONS

(J1) **Ali HeydariGorji**, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2022. Leveraging Computational Storage for Power-Efficient Distributed Data Analytics. *ACM Trans. Embed. Comput. Syst.*, 21, 6, Article 82 (November 2022), 36 pages. `https://doi.org/10.1145/3528577`.

(J2) Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, **Ali Heydarigorji**, Diego Souza, Brunno F˙ Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. 2020. Cost-effective, Energy-efficient, and Scalable Storage Computing for Large-scale AI Applications. *ACM Trans. Storage* 16, 4, Article 21 (November 2020), 37 pages. `https://doi.org/10.1145/3415580`.

(J3) Safavi, Seyede Mahya, Ninaz Valisharifabad, Robert Sabino, Hsinchung Chen, **Ali HeydariGorji**, Demi Tran, Jack Lin, Beth Lopour, and Pai H. Chou. “Analysis of cardiovascular changes caused by epileptic seizures in human photoplethysmogram signal.” arXiv preprint arXiv:1912.05083 (2019).

(J4) Torabzadehkashi, Mahdi, Siavash Rezaei, **Ali HeydariGorji**, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. “Computational storage: an efficient and scalable platform for big data and HPC applications.” *Journal of Big Data 6*, no. 1 (2019): 1-29. `https://doi.org/10.1186/s40537-019-0265-5`.

REFEREED CONFERENCE PUBLICATIONS

(C1) **A. HeydariGorji**, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves and P. H. Chou, "In-storage Processing of I/O Intensive Applications on Computational Storage Drives," 2022 23rd International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 2022, pp. 1-6, doi: 10.1109/ISQED54688.2022.9806270.

(C2) **A. HeydariGorji**, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves and P. H. Chou, “HyperTune: Dynamic Hyperparameter Tuning for Efficient Distribution of DNN Training Over Heterogeneous Systems,” *2020 IEEE/ACM International Conference On Computer Aided Design* (ICCAD), 2020, pp. 1-8.

(C3) **A. HeydariGorji**, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves and P. H. Chou, “Stannis: Low-Power Acceleration of DNN Training Using Computational Storage Devices,” *2020 57th ACM/IEEE Design Automation Conference* (DAC), 2020, pp. 1-6, `doi:10.1109/DAC18072.2020.9218687`.

(C4) M. Torabzadehkashi, **A. Heydarigorji**, S. Rezaei, H. Bobarshad, V. Alves and N. Bagherzadeh, “Accelerating HPC Applications Using Computational Storage Devices,” *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems* (HPCC/SmartCity/DSS), 2019, pp. 1878-1885.

(C5) Hsinchung Chen, Subramanian Meenakshi, **Ali HeydariGorji**, Seyede Mahya Safavi, Pai H. Chou, Cheng-Ting Lee, and Ruey-Kang Chang. "BlueBox: A Complete Recorder for Code-Blue Events in Hospitals." In *2019 International Symposium on VLSI Design, Automation and Test* (VLSI-DAT), pp. 1-4. IEEE, 2019.

(C6) M. Torabzadehkashi, S. Rezaei, **A. Heydarigorji**, H. Bobarshad, V. Alves and N. Bagherzadeh, "Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics," *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (PDP), 2019, pp. 430-437, `doi:10.1109/EMPDP.2019.8671589`.

(C7) Cheng-Ting Lee, Yun-Hao Liang, Pai H. Chou, **Ali Heydari Gorji**, Seyede Mahya Safavi, Wen-Chan Shih, and Wen-Tsuen Chen. 2018. "EcoMicro: A Miniature Self-Powered Inertial Sensor Node Based on Bluetooth Low Energy." In *Proceedings of the International Symposium on Low Power Electronics and Design* (ISLPED '18). Association for Computing Machinery, New York, NY, USA, Article 30, 1–6. `https://doi.org/10.1145/3218603.3218648`.

(C8) **A. Heydarigorji**, S. M. Safavi, C. T. Lee and P. H. Chou, "Head-mouse: A simple cursor controller based on optical measurement of head tilt," *2017 IEEE Signal Processing in Medicine and Biology Symposium* (SPMB), 2017, pp. 1-5, `doi:10.1109/SPMB.2017.8257058`.

(C9) S. M. Safavi, S. M. Sundaram, **A. Heydarigorji**, N. S. Udaiwal and P. H. Chou, "Application of infrared scanning of the neck muscles to control a cursor in human-computer interface," *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*(EMBC), 2017, pp. 787-790, doi: `10.1109/EMBC.2017.8036942`.

ABSTRACT OF THE DISSERTATION

Leveraging Computational Storage for Power-Efficient Distributed Data Analytics

By

Ali HeydariGorji

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2024

Professor Pai H. Chou, Chair

This thesis presents a family of computational storage drives (CSD) and demonstrates their performance and power improvements due to in-storage processing (ISP) when running big-data analytics applications. CSDs are an emerging class of solid-state drives (SSD) that are capable of running user code while minimizing data-transfer time and energy. Applications that can benefit from in-situ processing include distributed training, distributed inferencing, and databases. To achieve the full advantage of the proposed ISP architecture, we propose software solutions for workload balancing before and at runtime for training and inferencing applications. Other applications such as sharding-based databases can readily take advantage of our ISP structure without additional tooling. Experimental results on different capacity and form factors of CSDs show up to 3.1x speedup in processing while reducing the energy consumption and data transfer by up to 67% and 68%, respectively, compared to regular enterprise SSDs.

Chapter 1

Introduction

With the advent of data centers, which fuel the demand for virtually unlimited storage, it has been estimated that 2.5 exabytes of data is being created every day [2], from text to images, music, and videos. The deep learning revolution has given rise to running computationally intensive algorithms on these huge amounts of data. Different technologies have been developed to support this trend towards high storage and computation demands, and they shift the bottleneck as a result. This work investigates one of the opportunities made possible by this shift in bottlenecks, namely the elimination of data transfer from the storage unit to the host system for processing.

1.1 Motivation

The storage system where data originally resides plays a crucial role in the performance of data-intensive applications. To be processed, data always needs to be read from the storage units to the memory units of the application servers. As the size of the data increases, the role of the storage subsystem becomes more important. Computational Storage Drive (CSD) is a class of storage drives that can meet this growing demand by augmenting the storage drives with processing resources while eliminating unnecessary data transmission to the host's CPU. The main concept that distinguishes CSDs from conventional SSDs is in-storage processing (ISP) capability. ISP is the computing paradigm of moving computation to data storage as opposed to moving data to compute engines. ISP is considered by some as the ultimate form of near-data processing, as it must strike a balance between the storage and computing sides.

On the storage side, hard disk drives (HDDs) have been increasing in capacity with high reliability and very competitive cost. However, their performance has plateaued, and their power consumption is ultimately lower-bounded by mechanical motion. Solid state drives (SSD), which are based on NAND-flash memory and offer better performance at lower power consumption, have not been competitive in pricing until recently, but they are starting to find their way into data centers. The market share of enterprise SSDs jumped from almost 0% in 2010 to 12% in 2020 and 30% in 2024 [3, 4].

On the computing side, general-purpose graphic processing units (GPGPU) have turned conventional PCs into supercomputers in terms of their ability to execute parallel operations. However, GPUs are power-hungry and are not suitable for storage systems, which now are most concerned about the power budget, because power not only incurs significant cost, but cooling also incurs additional costs to operate and maintain and can become another source of unreliability. As a result, GPGPUs are not so suitable for in-storage processing. Instead, they often assume data to be on the local computer. However, if the data resides on a storage system rather than the memory, the communication network will become the bottleneck, as data from the storage to the processor consumes 5000x more energy and 2500x more latency compared to loading it from the volatile memory [5].

One growing class of applications in wide use is data analytics, including artificial intelligence and search engines. In these applications, the processing nodes require huge volumes of raw data, and the transfer of raw data can easily become the bottleneck in conventional architectures with separate host and storage. In-storage processing can minimize or eliminate this bottleneck by moving part or

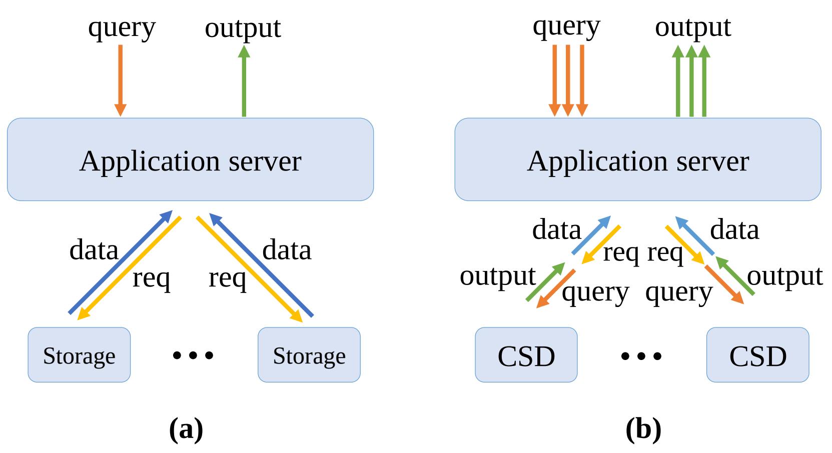


Figure 1.1: Handling queries on (a) a server with generic storage systems vs. (b) a server with CSDs.

all the computation to the storage unit and sending only the output back to the host. To support ISP, the storage system needs to be capable of not only storage but also processing, and computational storage drives (CSD) are being used to build servers such as that shown in Fig. 1.1.

1.2 Contributions

Our goal in this work is to enable computational storage drives to operate as a stand-alone near-data processing engine that can run distributed processing on a cluster of CSDs. To support in-storage processing, we have developed a custom software stack that enables CSDs to run a full-fledged operating system (e.g., Linux) and to provide seamless access to data stored on the flash, which in turn enables application developers to run general-purpose application binaries on the storage drive without modification and to access the data residing on the flash. To support efficient communication with the ISP engine, we have developed a TCP/IP-based tunneling system that is implemented on the regular NVMe/PCIe bus and allows the ISP engine to communicate with other processing nodes and the Internet. To enable running distributed processing of deep neural network training applications on a cluster of CSDs, we have developed Stannis, a framework to efficiently distribute DNN training over heterogeneous nodes. To make our distributed processing setup more robust to external workloads, we proposed HyperTune, a run-time mechanism incorporated into Stannis to ensure that all processing nodes experience minimum stalling during neural network training. Finally, we developed a scheduler to run general-purpose applications such as database queries on our system. Here is a summary of contributions presented in this work:

- We have developed a software stack that enables us to run an operating system inside our computational storage drive.
- We have developed a TCP/IP-based tunneling system on NVMe/PCIe bus to communicate between the ISP engine and other processing nodes and also the Internet.
- We have developed a framework to efficiently distribute various applications such as DNN training over a cluster of CSDs.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides background and related work on the newly emerged data-transfer bottleneck, the concept of computational storage, computational storage drives proposed to date, and their use in heterogeneous systems. Chapter 3 describes the proposed CSD architecture and its design, including both hardware and software. Chapter 4 explains our developed application for efficient and robust distribution of deep neural network training over heterogeneous systems, and Chapter 5 covers the other classes of applications that we enhanced and deployed on CSDs. Chapter 6 evaluates our approach with experimental results on our CSD design. Finally, Chapter 7 concludes this dissertation with directions for future work.

Chapter 2

Related Work

In conventional processing systems, data needs to be transferred closer to the processing unit, typically a high-speed, volatile memory. This transfer of data can become a communication bottleneck, preventing the system from getting its maximum utilization. In this chapter, we examine the common issue of data transfer bottlenecks and how near-data processing, or in this case in storage processing, can help remove such bottlenecks. We then go over different designs for computational storage units and why we chose Computational Storage Drives(CSD) approach as our proposed solution. Finally, we will go over different approaches to designing CSD.

2.1 Data Processing Systems

In this section, we conduct an in-depth analysis of the prominent technologies utilized in storage units and their integral role within processing systems. Moreover, we investigate the various challenges that these technologies encounter as a direct result of the increasing size and quantity of data necessitating efficient processing.

2.1.1 Storage in processing systems

The storage system where data originally reside plays a crucial role in the performance of applications. In a cluster, the data should be read from the storage system to the memory units of the application servers to be processed. As the size of data increases, the role of the storage system becomes more important, since the nodes need to talk to the storage units more frequently to fetch data and write back the results. Recently, cluster architects have considered solid-state drives (SSDs) over hard disk drives (HDDs) as the primary storage units in modern clusters due to better power efficiency and higher data transfer rate [6].

Solid-State Drives (SSDs) utilize NAND flash memory as the underlying storage medium, which offers superior speed and power efficiency compared to the magnetic disks used in Hard Disk Drives (HDDs). Consequently, SSDs are recognized as more efficient alternatives to HDDs. While SSDs eliminate the complexities associated with physical disk management, timing, seek and rotation mechanisms, spin up/spin down procedures, head protection, data consolidation from multiple platters, signal processing, and error correction, they introduce new challenges including wear leveling, block erasing, mapping strategies, and garbage collection. To effectively manage the flash memory array, SSD controllers employ multi-core processors. Additionally, SSDs commonly feature high-speed interfaces such as NVMe over PCIe for efficient communication with the host. The incorporation of such interfaces necessitates the inclusion of increased processing power within SSDs. A contemporary SSD controller comprises two principal components: 1) a front-end processing engine responsible for facilitating high-speed host interface protocols such as NVMe/PCIe [7], and 2) a back-end processing engine tasked with executing flash management routines. These two engines communicate with each other to successfully execute the input/output (I/O) commands received from the host.

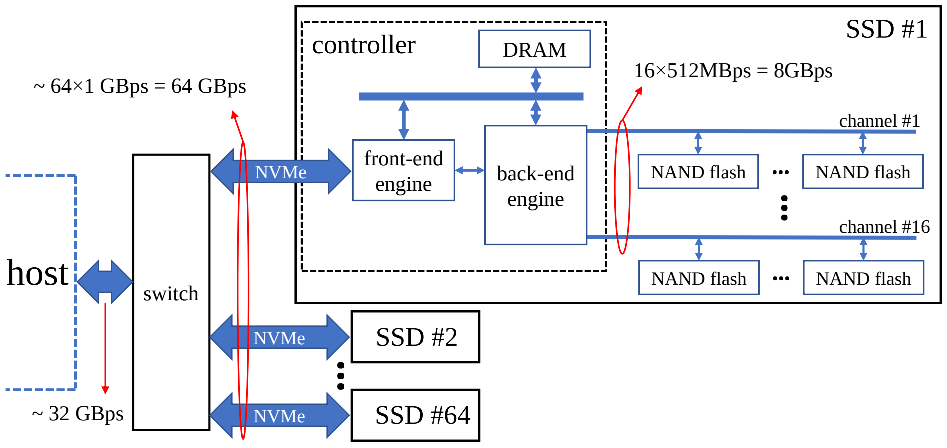


Figure 2.1: An Example of Modern SSD architecture

2.1.2 Scaling up Storage Systems

Webscale data center designers have been developing storage architectures that favor high-capacity hosts, and this fact is highlighted at OpenCompute (OCP) by Microsoft Azure and Facebook calls for up to 64 SSDs attached to each host [8]. Fig. 2.1 shows such a storage system with 64 SSDs attached to the host. For the sake of simplicity, only the details of one SSD are demonstrated. 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. This huge bandwidth decreases to about 1 GBps due to the complexity of the host interface software and hardware architecture. In other words, the accumulated bandwidth of all internal channels of the 64 SSDs reaches the product of the number of SSDs by the number of channels per SSD by 512 MBps (bandwidth of each channel), which equals to 512 GBps. While the accumulated bandwidth of the SSDs' external interfaces is equal to 64 (the number of SSDs) multiplied by 1 GBps (the host interface bandwidth for each SSD) which is 64 GBps. This bandwidth further decreases as SSDs connect to the host through a PCIe switch, which typically has a standard bandwidth of 32 GBps.

Overall, the interface to the host pays a 16x bandwidth penalty compared to the aggregated internal bandwidth of all SSDs. That is, if the internal components of the SSDs can read the entire 32 TB of data in about one minute, the host would take 16 minutes. Additionally, in such storage systems, data need to continuously move through the complex hardware and software stack between hosts and storage units, which consume considerable energy and can dramatically decrease the energy efficiency of large data centers. Hence, storage architects need to develop techniques to decrease data movement, and in-storage processing (ISP) technology has been proposed to overcome the aforementioned challenges by bringing the process closer to data.

2.1.3 Traditional Processing Pipeline

In the traditional CPU-centric scheme, data always move from storage devices to the CPU for processing, and this *von Neumann bottleneck* is the root cause of the challenges above, especially when many SSDs are connected to the host. ISP is a contrary approach based on the idea of "*bringing the process to data*" to its ultimate boundaries, where a processing engine inside the storage unit takes advantage of high-bandwidth and low-power internal data links and processes data in place. In fact, "*bringing the process to data*" is the concept that leads to the emergence of both ISP and distributed processing platforms such as Hadoop[9]. Later in this section, we describe how the Hadoop platform and ISP can simultaneously work together in a cluster.

ISP minimizes the data movements in a cluster and also increases the processing horsepower of the cluster by augmenting the whole system with power-efficient processing engines. In fact, ISP is applicable to both HDDs and SSDs, although modern SSD architectures provide better tools for developing such technologies. Those SSDs that can run user applications in-place are called *computational storage drives* (CSDs). These storage units are augmentable processing resources, which means they are not designed to replace the high-end processors of modern servers, but instead, they can collaborate with the host's CPU by providing their efficient processing horsepower to the system.

Note that CSDs are fundamentally different from object-based storage systems, such as Seagate

Kinetic HDDs [10], which transfer data at the object level instead of the block level. The object-based storage units can receive objects (i.e., image files) from the host, store them, and at a later time, retrieve the objects back to the host using the object IDs. Consequently, the host does not need to maintain metadata of the block addresses of the object. On the other hand, CSDs can run user applications in-place without sending data to the host. There is a vast literature in this field that proposes different CSD architectures and investigates the benefits and challenges of deploying CSDs for running applications in-place, as to be reviewed in the Related Work section.

A while ago, when the cost of data movement was insignificant compared to the computational cost, a centralized storage system was effective, as it also allowed other hosts to send requests to it for fetching or writing data blocks. However, today's data-intensive applications require large data bandwidth to the storage system, and such huge data movements drastically increase energy consumption. With the emergence of big data, centralized storage is no longer viable, and the traditional approaches fall short of satisfying the demands of super-scale applications, which call for scalable processing platforms. To answer these demands, distributed processing platforms such as Hadoop are proposed to process data near where they reside [11].

2.1.4 Data Transfer Bottleneck

The trend towards SSDs has also brought major changes in the architecture of storage area networks (SAN). In traditional SANs, multiple HDDs are attached to the server via SCSI or SATA, and the servers are connected to each other via a high-speed protocol such as Fibre Channel (FC). Although SATA with its 750 MBps bandwidth sufficed for the data transfer rate of HDDs, SATA cannot keep up with the performance that can be extracted from contemporary NAND flash media. Newer industry-standard protocols such as NVMe have resolved this bottleneck with not only a much higher transfer rate but also by connecting to the CPU more directly via PCIe. However, even such advances still cannot remove those bottlenecks on the long path from the non-volatile data storage (flash) to the processing units (CPU, GPU), which continue hindering the full utilization of data read/write speed in storage servers.

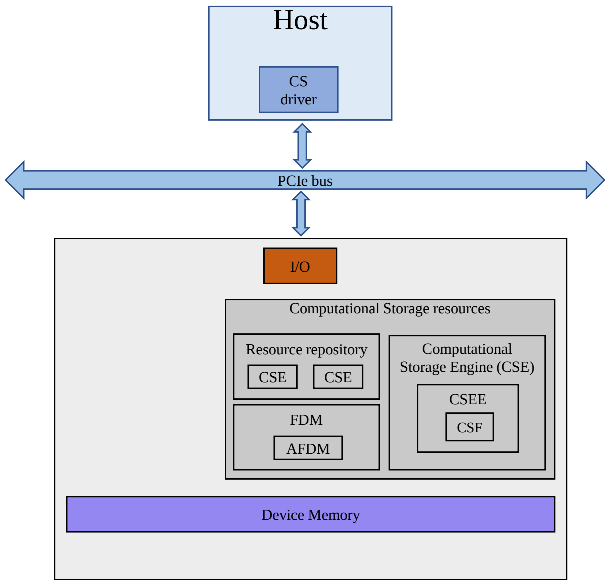
In I/O-intensive applications such as DNN training or NLP, this bandwidth mismatch can easily become the bottleneck of the entire operation. For example, the datasets used for training deep neural networks are larger than the DRAM size of the computing systems by tens of orders of magnitude; the embedding tables of an NLP application such as a recommendation engine can exceed tens of gigabytes, much more than the typical DRAM capacity of such systems. Studies show that even by using efficient methods such as pipelining and prefetching, up to 70% of the training time for DNNs can be wasted on blocking I/O operations that bring the raw, unprocessed training data to the processor. These challenges signify the importance of near-data or in-storage processing.

2.2 Computational Storage

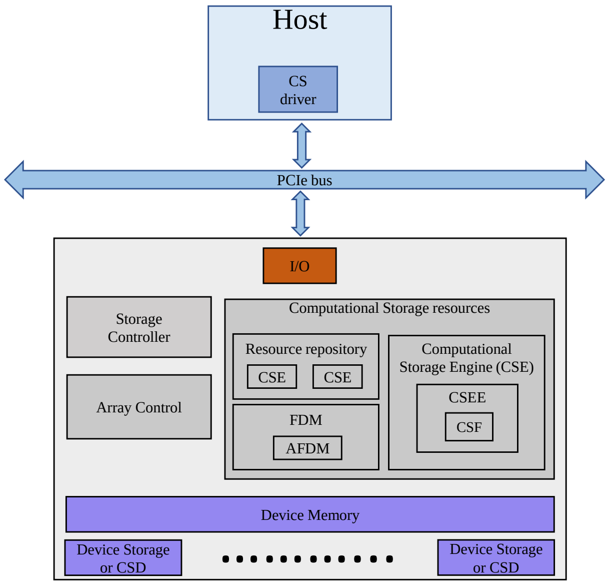
Computational storage architectures enable improvements in application performance or infrastructure efficiency or both through the integration of compute resources outside the traditional compute-and-memory architecture. The computing resources may be either directly integrated with storage or between the host and the storage. The goal of these architectures is to enable parallel computation while alleviating constraints on existing computing, memory, storage, and I/O resources. Fig. 2.2 shows different classes of computational storage, or CSx [12], where x may be processor, drive, or array.

2.2.1 Computational Storage Processor (CSP)

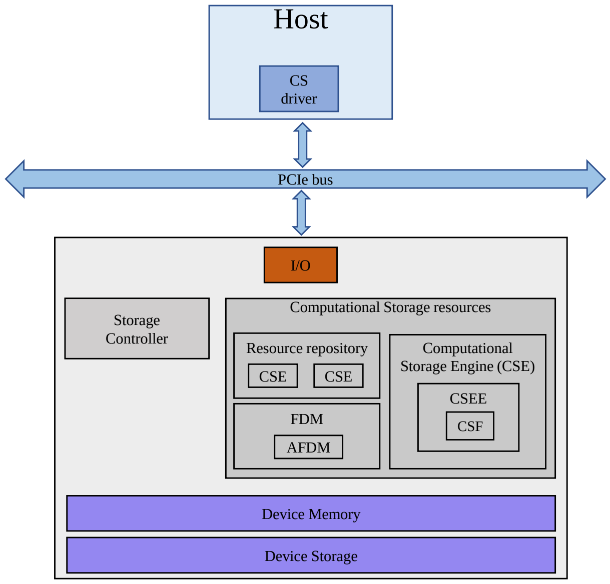
A CSP is a component capable of executing one or more Computational Storage Functions (CSF) for an associated storage system *without providing persistent data storage*. The CSP contains



Computational Storage Processor



Computational Storage Array



Computational Storage Drive

Figure 2.2: Classes of Computational Storage Systems

computational storage resources (CSR) and *device memory*. The mechanism by which the CSP is associated with the storage system is implementation-specific.

2.2.2 Computational Storage Drive (CSD)

A CSD is a component that can execute one or more CSFs *while providing persistent data storage*. The CSD contains a storage controller, CSR, device memory, and persistent data storage. A CSD may continue to function as a standard storage drive with existing host interfaces and drive functions. As such, the system can have a storage controller with associated storage memory along with storage addressable by the host through standard management and I/O interfaces.

2.2.3 Computational Storage Array (CSA)

A CSA is a storage array capable of executing one or more CSFs. As a storage array, a CSA contains control software, which provides virtualization to storage services, storage devices, and CSRs for the purpose of aggregating, hiding complexity, or adding new capabilities to lower-level storage resources. The CSRs in the CSA may be centralized or distributed across CSDs or CSPs within the array.

All CSx classes are similar in design as they all consist of the following components:

Computational storage resources (CSR), which contain a *resource repository* to store blocks such as *computational storage functions* (CSFs) and/or *computational storage engine environments* (CSEEs), a *function data memory* (FDM) that can be partitioned into *allocated function data memory* (AFDM), and one or more *computational storage engines* (CSEs),

a storage controller (for CSD) or an Array Controller (for CSA),

a device memory , and

a device storage (only for CSD and CSA).

Of these three classes, CSD is the most efficient approach for several reasons. First, it has a shorter data path than CSP does. Second, it has finer granularity and fewer complications compared to the CSA.

2.3 Computational Storage Drives

Computational storage drives can be structured in different ways. The computing resource may use the existing SSD controller, a separate processing engine, or an integrated SSD+ISP engine. We also discuss issues with accelerators such as FPGAs or GPUs.

2.3.1 Utilizing SSD Controller's Processing Resources

The easiest way to develop a CSD is to use the processing engines already available in the SSD [13, 14, 15, 16, 17, 18]. This subsection reviews these CSD implementations.

Lee et al. [14] implements a CSD based on the Jasmine OpenSSD Platform [19] to perform external sorting. Since the platform has a single ARM7TDMI-S core running at up to 87.5 MHz, they use the same processor for ISP, which yields a low performance improvement of 39%, compared with the traditional external sorting algorithm.

RecSSD [17] proposes a near-data processing solution that improves the performance of the underlying SSD storage for embedding table operations for recommendation applications. It utilizes the internal SSD bandwidth and reduces data communication overhead between the host processor and SSD by offloading the entire embedding table operation to the SSDs, including gather and aggregation computations. The hardware is the commercial Cosmos+ OpenSSD evaluation platform [20] with a dual 1-GHz ARM Cortex-A9 core featuring a custom software stack on the flash translation layer (FTL). RecSSD reduces end-to-end neural recommendation inference latency by 4x compared to off-the-shelf SSD systems at the cost of losing SSD performance, since the ISP engine shares the same processing cores with the conventional SSD controller.

Biscuit [18] is a near-data processing framework for running applications distributed both on the host system and the storage device. Biscuit uses two ARM R7 cores that are originally dedicated for running conventional SSD controlling tasks. By offloading some parts of the computation to the SSD, the overall performance improves. The original read/write functionality of an SSD cannot be suspended because of running the user application in-place. Just like previous projects, it is unclear how much the performance of Biscuit degrades in the case of simultaneous host I/O requests and running user applications in-place. They also propose an innovative flow-based programming model to offload tasks to the embedded processing engine of Biscuit dynamically, but designing complex user applications based on the programming model is potentially time-consuming.

2.3.2 Coupling an External Processing Engine

The second approach is to couple an external processing engine with the storage drive [21, 22, 23, 24]. This subsection reviews these representative works.

Jun et al. [21] proposes a scalable architecture, code-named BlueDBM, that is a NAND-based storage system coupled with an external FPGA accelerator. It shows up to 10-fold speedup for running predefined tasks such as nearest-neighbor search or graph traversal, compared to a system using no CSD. However, such systems have a drawback in design and implementation time, and reconfigurability. Deploying an FPGA-based unit requires RTL (register-transfer level) design, which significantly complicates the deployment of new tasks [25, 26]. Likewise, reconfiguring an FPGA requires loading a compressed bitstream into the configuration memory, which can take hundreds of milliseconds [27, 28, 29].

Our early prototype, CompStor, also followed the same concept, and there was an off-chip ISP engine attached to the main FPGA-based flash storage controller. Such a design may suffer from low efficiency due to constantly transferring the data off-chip to the ISP engine [24].

Several research works have implemented data analytics on SmartSSD, a well-known CSD platform that uses external FPGA as an ISP engine [30, 23, 22, 31, 32]. SmartSSD is a 4-TB NVMe storage drive with an external FPGA as the ISP engine. Chapman et al. [23] investigated a computational storage platform based on SmartSSD for big-data analytics. It can directly communicate with the storage part by transferring data in P2P mode. With their software stack, it runs user binaries without modification, making it easy to port applications. The evaluation shows that this system can achieve 6x improvement in query times and an average of 60% lower CPU usage. However, no power and energy results are reported, even though the FPGAs and the off-chip accelerator are likely to be power-hungry in this setup. Just like other research works, FPGA-based ISP requires extensive efforts to implement new applications. Moreover, this design does not represent a true in-storage processing platform, as the data has to migrate from the SSD drive to the ISP module. Hence, the data transfer bottleneck still persists in high-speed applications.

2.3.3 An Integrated SSD + ISP Engine

The last approach is to build an integrated ISP+SSD controller to eliminate off-chip communication for in-storage processing. To the best of our knowledge, there are only two designs that incorporate such integration: ScaleFlux and Catalina.

ScaleFlux [33] is an FPGA-based CSD where both the SSD controller and the ISP engine are implemented on an FPGA. Like other FPGA-based ISPs, the critical functions that need to be accelerated must be implemented in RTL and off-loaded onto the FPGA.

Catalina [34] is a CSD in AIC form factor that uses Xilinx's Virtex UltraScale+ SoC to implement

Table 2.1: Technical Specifications of commercial CSDs.

Name	Capacity	SSD controller	CS resource	ISP engine	Support for Operating systems	Programming model
OpenSSD Jasmine[19]	64 GB	ARM7TDMI-S	Shared	ARM7TDMI-S	No	bare-metal software
OpenSSD Cosmos+[20]	512 GB	XC7Z045-3FFG900 Zynq-7000 FPGA	Shared	Dual core ARM Cortex-A9 and dual Neon DSP Co-processor	No	RTL design and bare-metal software
Samsung SmartSSD[30]	4 TB	ASIC	Dedicated	Kintex XCKU15P UltraScale+ FPGA	No	RTL design
ScaleFlux[33]	8 TB	FPGA	Dedicated	FPGA	No	RTL design
Newport	32 TB	ASIC	Dedicated	Quad core ARM A53 processor and four Neon DSP Co-processor	Ubuntu and Debian	Software on Linux OS and TCP/IP networking

the ISP and SSD controller functionality. Unlike ScaleFlux, Catalina uses two ARM Cortex R5 embedded real-time processors and FPGA programmable resources for the SSD controller and implements the ISP engine on a quad-core ARM A53 application processor.

Our proposed solution, called Newport, takes in-storage processing a step further by utilizing a standalone ASIC to include both the SSD controller and a dedicated ARM processor for running general applications. To the best of our knowledge, Newport is the first and only single ASIC-based computational storage drive controller with a multi-purpose processing engine. The ARM processor can run Linux executables without source-code modification or recompiling. We have also implemented a TCP/IP-based tunneling system that allows the in-storage processing engine to connect to a network, including the Internet and other processing nodes. Table 2.1 summarizes the technical characteristics of some of the well-known CSDs. The next section details the hardware and software that enable our system to function both as a regular storage system and as a high-capacity standalone processing node.

2.4 CSDs in Heterogeneous Computing Architectures

CSD is a new concept that is making its way into the data-storage market. In some ways, CSD resembles GPU in that both are considered hardware accelerators that communicate with the main processor through PCIe. Due to the long expected lifetime, high requirements for quality of service (QoS), and limitations in working conditions (available space, temperature, etc.) in storage servers, a suitable replacement for conventional SSDs in enterprise-level storage servers should fit in the same power, size, and working-condition envelope. These constraints eliminate alternatives such as GPU, TPU, or FPGAs to act as an accelerator in storage servers. However, what differentiates CSDs are the ultra-low power footprint and very small computation overhead size that makes them indistinguishable from a conventional SSD in terms of power consumption and regular operation. Besides, sitting behind the (relatively) slow front-end interface, the ISP can have much higher-speed access to the stored data on the flash, making it suitable and more efficient than the host to perform low-computation, I/O-intensive applications.

Compared to FPGA platforms, our CSD design benefits from a central ASIC that hosts both the SSD controller and the ISP engine. Although FPGA is more flexible than ASIC and can massively parallelize an architecture, they consume considerably more power, require more silicon area, and can work with lower frequencies and throughput [35, 36]. An FPGA platform is ideal for rapid prototyping and engineering development, but it can never go beyond a mass-production level.

To summarize, CSD is not meant to compete with high-end GPUs or FPGAs. Instead, it introduces a new paradigm that enhances the performance of storage systems by augmenting the host with extra processing power at the cost of minimal overhead and by running the I/O-intensive parts of the process with fewer data transmissions. A comparison is shown in Fig. 2.3.

2.5 Cost Analysis

Compared with an enterprise-grade SSD based on a conventional controller, a CSD is expected to contain additional processing resources such as a more powerful processor to efficiently run a variety of software applications. Some types of CSD make use of FPGA-based hardware accelerators. It is also fair to assume that CSDs would require larger DDR memory (for the application computation) than the conventional SSD does.

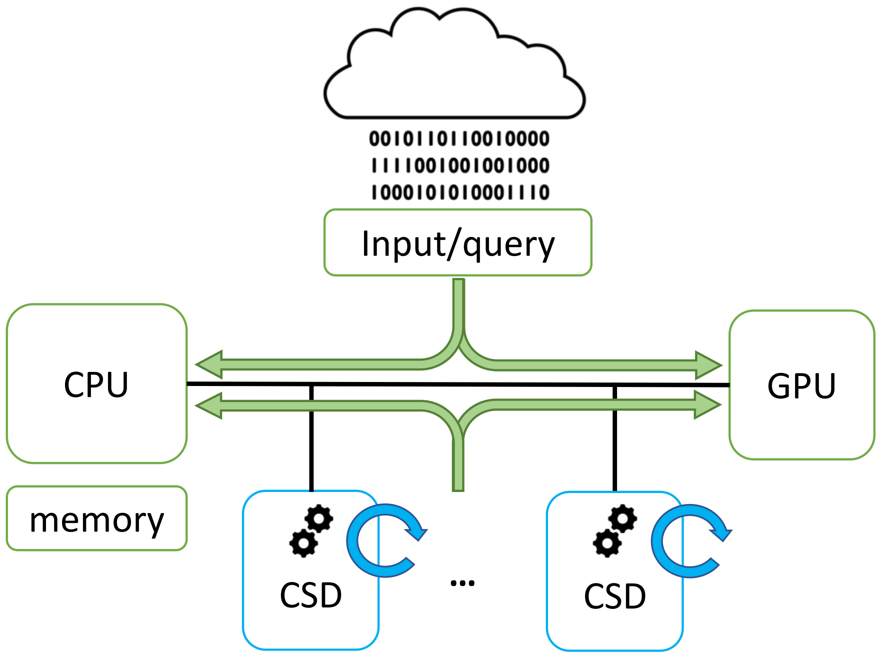


Figure 2.3: CSD as a complementary component in a high-performance architecture.

Interestingly, the SSD bill of material (BOM) analysis [37, 38] shows that the main cost difference between an SSD and a CSD would be marginal, given that the current SSD price is largely dominated by the cost of NAND flash chips. The exception is for FPGA-based CSD implementations due to the relatively high cost of FPGA devices. For example, in 2020, an enterprise-level 8-TB NVMe SSD is priced starting at around $1,600 in the open marketplace, and the spot price of NAND flash TLC 512-Gb chips is approximately $0.15 per GB at the time of writing this paper[1]. In 2024, the same configuration cost is down to $0.04 per GB. Other costs such as DRAM, miscellaneous components, and manufacturing costs can account for 10% to 25% of the CSD price. The amount of additional memory will vary for different designs, but again, it is not expected to be a significant factor in the total cost.

In this section, we analyze the cost of the most common and prevalent types of commercial CSDs as shown in Fig. 2.4. These types of CSD designs have been actively discussed at a technical

[1]While the prices of an SSD and NAND flash devices have wide variability at any time and more over time, the analysis provides a good big picture of the economic feasibility of CSDs. Even though NAND flash price per GB has been continually dropping, both NAND flash density and average SSD capacity have been increasing steadily, therefore, compensating for it when considering the cost for a CSD unit. source: *https://www.dramexchange.com*

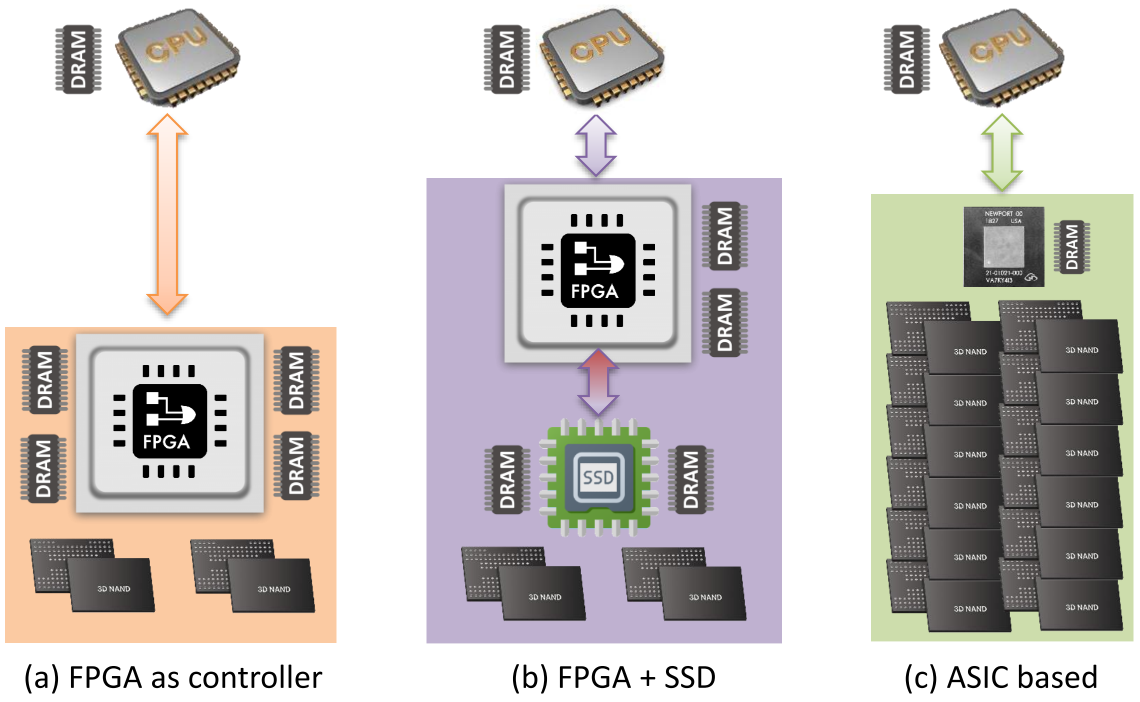


Figure 2.4: Commercial CSDs implementation approaches.

working group of the Storage Networking Industry Association (SNIA) [39], which was set up in the year of 2019 to standardize device interoperability, management, and security among SSD vendors developing a number of different technologies and approaches to CSDs. Fig. 2.4.a depicts the FPGA-based controller approach [33]. This design provides significant flexibility thanks to the re-programmable logic at the expense of a convoluted programming model and higher cost. The second variant, labeled *FPGA + SSD* [40, 41, 33] is depicted in Fig. 2.4.b. [42] presents very similar drawbacks from the cost point of view. Finally, Fig. 2.4.c depicts the ASIC-based architecture represented by the Newport CSD [43].

Table 2.2 shows the estimated cost for low and high volumes of a mid-range capacity 8-TB NVMe CSD. The costs of the main components (DRAM, FPGA, off-the-shelf SSD controller, and NAND flash) are obtained from Website [44, 45]. When one excludes NAND flash from the BOM cost as shown in Fig. 2.5, the ASIC-based approach turns out to be approximately 4x less costly.

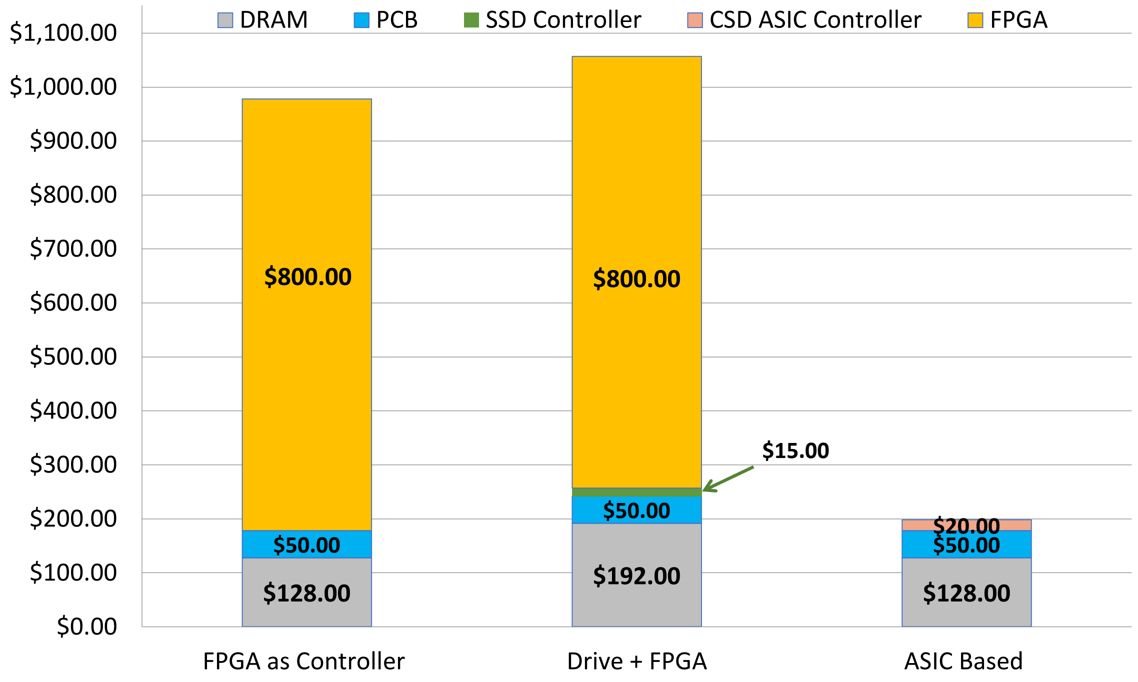


Figure 2.5: 8TB Commercial CSDs relative cost - NAND flash excluded.

Table 2.2: 8TB Commercial CSDs cost analysis - high volume

Components	**FPGA as Controller**	**Drive + FPGA**	**ASIC Based**
FPGA	$800	$800	-
8TB NAND	$816	$816	$816
DRAM	$128	$192	$128
Passive Components	$75	$75	$75
PCB	$50	$50	$50
CSD ASIC Controller	-	-	$20
SSD Controller	-	$15	-
Power Loss Capacitors	$5	$5	$5
TOTAL	$1,874	$1,953	$1,094

2.6 Summary

This chapter discusses data-processing systems and the role of storage in their performance. As the size of data increases, the storage system becomes more important, and scaling up storage systems becomes more challenging. One solution for these challenges is ISP technology, which involves a processing engine inside storage units that can process data in-place, which in return, minimizes data movements in a cluster and increases the processing horsepower of the cluster. ISP can be implemented using Computational Storage Drives (CSDs), which are SSDs that can run user applications in-place and collaborate with the host's CPU to augment their efficient processing horsepower to the system. While utilizing the existing SSD controller is the easiest way to develop a CSD, coupling an external processing engine provides greater performance gains. However, both these approaches require off-chip communication for in-storage processing. The third approach, an integrated SSD+ISP engine, eliminates this requirement and is more efficient for high-speed applications. CSD is a suitable replacement for conventional SSDs in enterprise-level storage servers due to its ultra low-power footprint and very small overhead size, making it indistinguishable from a regular SSD. CSD introduces a new paradigm that enhances the performance of processing systems by running the I/O-intensive parts of the process within storage drives, resulting in less data transmission. The cost difference between an SSD and a CSD would be marginal, given that the current SSD price is largely dominated by the NAND flash chips. The exception is for FPGA-based CSD implementations due to the comparatively high cost of FPGA devices. Analyzing the cost of the most common and prevalent types of commercial CSDs shows that the ASIC-based approach turns out to be approximately 4x less costly when excluding NAND flash from the BOM cost.

Chapter 3

Design of a Computational Storage Drive

This chapter demonstrates the different stages of development of the Newport family CSDs and how the early off-chip CSD architecture led to a more mature and complete solution. First, we describe the fundamentals of a general SSD architecture and how the computational storage concepts can fit in this domain. Then, we present our early CSD architecture and discuss the shortcoming of this off-chip design. In the end, we explain the detailed hardware and software architecture of the Newport family of CSDs.

3.1 Solid State Drive

Solid-state drives (SSDs) are becoming popular both in personal computers and enterprise-level storage systems. They deliver significantly higher performance than traditional hard disk drives (HDDs) do. In addition, due to the lack of mechanical components, SSDs can be smaller in size and consume less power while accommodating much higher capacity. A modern SSD is composed of two main components: the SSD controller and the non-volatile storage media [46]. The controller unit can be further broken down into the front-end, back-end, and central controlling units.

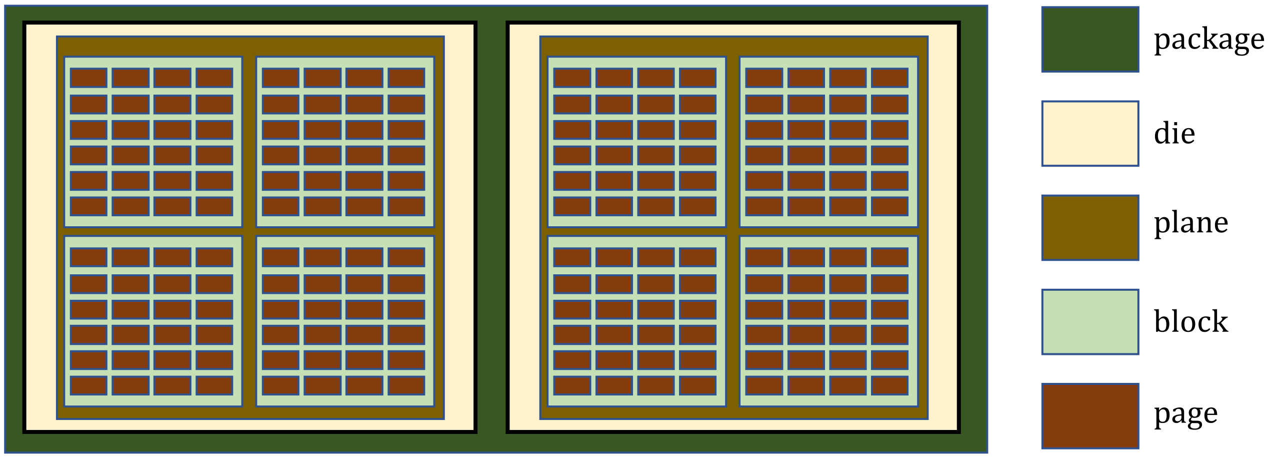


Figure 3.1: Flash chip organization in a solid state drive.

3.1.1 NAND Flash Media

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, each of which contains multiple blocks, where a *block* is the smallest erasure unit. Inside each block are multiple *pages*, which are the smallest programming (writing) units. The key point in this hierarchical architecture is the programmable unit versus the erasable unit. The NAND flash memory can be programmed at the page level, usually, 4–16 KB, while the erase operation cannot be done on a smaller segment than a block, which can be several MB. While the cost of SSD is higher than that of HDDs, the difference is rapidly shrinking, thanks to new flash technologies such as Quadruple-Level Cell (QLC) flash and the upcoming Penta-Level Cell (PLC) flash at the cost of increased bit error rate, i.e., lower reliability. Such advancements have paved the way for the adoption of NAND flash-based storage technology everywhere, from consumer goods to the cloud and the edge [47, 48, 49].

3.1.2 Controller

As the brain of the SSD, the controller runs various functions to efficiently manage the NAND flash media. Examples of such functions include garbage collection, which reclaims blocks containing invalid data, and wear leveling, which ensures that flash blocks are erased and written evenly to prolong the SSD life. A Flash Translation Layer (FTL) maps logical addresses to physical addresses so that the logical view of storage is separated from the inner management of the physical memory. A major responsibility of the controller is to handle data writes. The data cannot be overwritten on flash memory and may 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 of data, update the page content, and write it back to an erased block. To modify the write operations, the garbage collector routine erases the blocks during off-peak times to maintain optimal write speeds [50].

3.1.3 Front-end Interface

Protocols for transferring data between the host and the storage devices include SATA [51], SAS [52], and NVMe over PCIe [53]. SSD drives can execute multiple I/O commands simultaneously and with low latency. Among the interfaces above, NVMe is more suited for enterprise SSDs and has impressive performance in transferring data between the SSD drives and the host. This makes NVMe the protocol of choice for our proposed CSD architectures. The technical specification of NVMe over PCIe protocol is quite complicated and is outside the scope of this research. The rest of this section reviews the NVMe protocol.

Peripheral Component Interconnect express (PCIe) [54] is a high-speed bus standard that uses a set of unidirectional pairs of serial and point-to-point links, called lanes. A PCIe slot can have 1, 4, 8, or 16 lanes, denoted as x1, x4, x8, and x16, respectively. The PCIe protocol 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 protocol. Each lane of PCIe Gen 1, Gen 2, Gen 3, and Gen 4 provides a data bandwidth of 250, 500, 985, and 1970 MBps, respectively. PCIe links can be used to connect different peripherals to hosts, such as video cards, expansion cards, and storage units. Our proposed CSD architectures contain a host interface based on PCIe Gen 3 x4, which can provide a bandwidth of up to 3940 MBps.

NVMe protocol uses the PCIe data link to transfer data between a host and an SSD. Traditional data transfer protocols were developed for HDDs that have just one queue for submitting I/O commands, but they cannot sustain SSDs' much higher bandwidth. To support SSDs' ability to run multiple I/O commands at the same time, MVMe provides up to 64K data transmission queues, each of which supports up to 64K parallel I/O commands.

3.1.4 Standard Form Factors

Several industry-standard form factors for flash-based storage devices exist. While some target commodity machines, others are designed for data centers and edge infrastructure [55].

Among all form factors, EDSFF, for *Enterprise and Data Center SSD Form Factor*, is the gold standard that also takes into account the thermal condition and the harsh environment of enterprise systems. It offers a set of flexible specifications of form factors, including E1.L, E1.S, E3.L, E3.S, etc. The proposed CSD, Solana, is designed and manufactured for edge systems, and the E1.S form factor has been chosen to address thermal challenges and also to provide enough room for a large number of NAND flash chips. The width and length of Solana are 31.5 and 111.49 millimeters, respectively. Moreover, a heat sink can be mounted on drives in E1.S form factor for passive thermal dissipation.

The U.2 form factor is 2.5-inch and the most common for SSDs, offered in PCIe (with NVMe), SAS, or SATA interfaces. Based on capacity and interface, the 69.85 mm $\times$ 100 mm U.2 drive can have a thickness of 7 mm or 15 mm. U.2 can be deployed in a wide range of systems, from laptops and desktops to enterprise storage servers. U.2 is defined as compliance with the PCI Express SFF-8639 Module specification, instead of referencing SAS or SATA SSDs as was typically done previously.

The M.2, formerly known as *Next Generation Form Factor* (NGFF), is widely used for internally mounted SSDs. It supports PCIe, SATA, and USB interfaces and comes in various widths and lengths. It also has keying notches on the edge connector to designate various interface or PCIe lane configurations. M.2 is smaller than the typical 2.5" SSD and is typically removable.

We used our Newport controller to prototype CSD in all three form factors. Our M.2 CSD is a Gen 3 x8 storage device with a capacity of up to 8 TB and 6 GB of onboard DRAM. Note that due to space limitations, the size of DRAM is smaller compared to the U.2 form factor. The CSD in E1.S form factor is similar to that in M.2, but with more PCB area and power range. It has 12 TB of NAND flash and 6 GB of DRAM. Table 3.1 shows the specifications of our CSDs in different form factors, and Fig. 3.4 shows the actual prototypes.

3.2 CompStor: Early off-chip CSD Solution

The first CSD that we designed and prototyped was CompStor [24] ("computational storage"), the first with a dedicated quad-core application processor as the ISP engine. In this early CSD prototype, we chose to separate the development of the conventional flash-management functionality from the innovative ISP capabilities to avoid potential errors and extra complications. Therefore, CompStor is composed of two subsystems, namely *SSD subsystem* and *ISP engine*, on separate boards that implement the flash management routines and ISP engine, respectively.

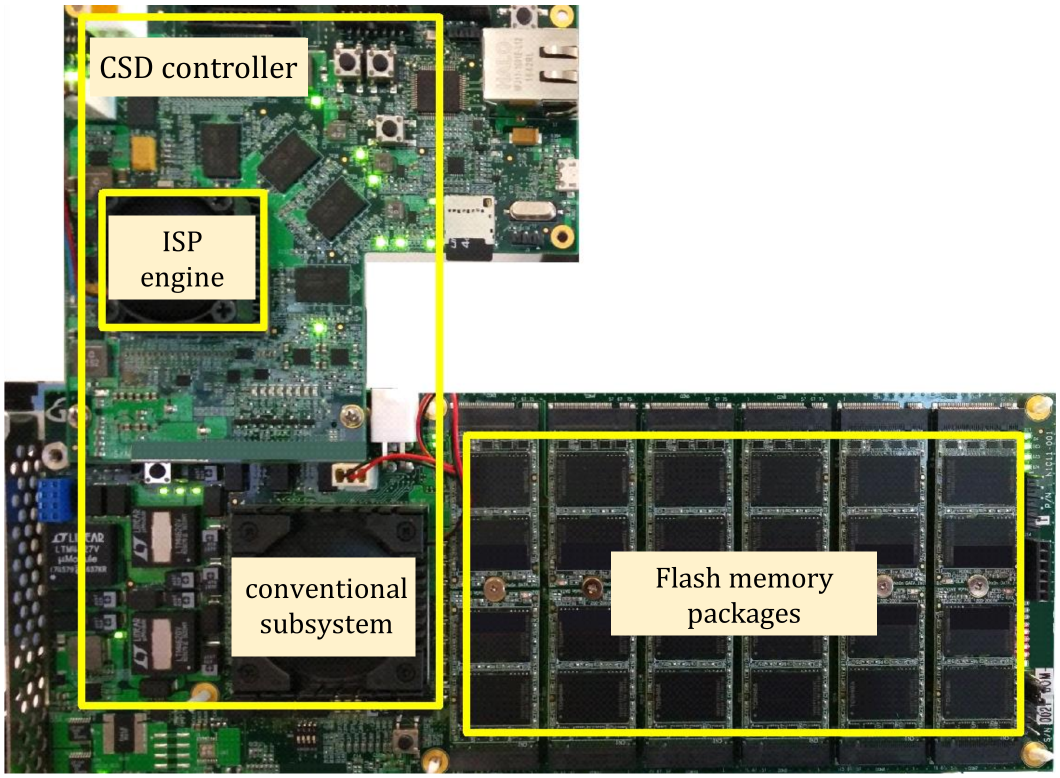


Figure 3.2: CompStor prototype.

3.2.1 SSD Subsystem

The *conventional subsystem* contains a controller, 2 GB of DRAM, and an array of flash packages. The controller is composed of two MicroBlaze processors [56], 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.

An internal data bus in the *conventional subsystem* transfers data between different components. This data bus is attached to the 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 storage unit with ISP capabilities. There is an FPGA mezzanine card (FMC) connector [57] that provides the connection between these two subsystems. In other words, CompStor is an

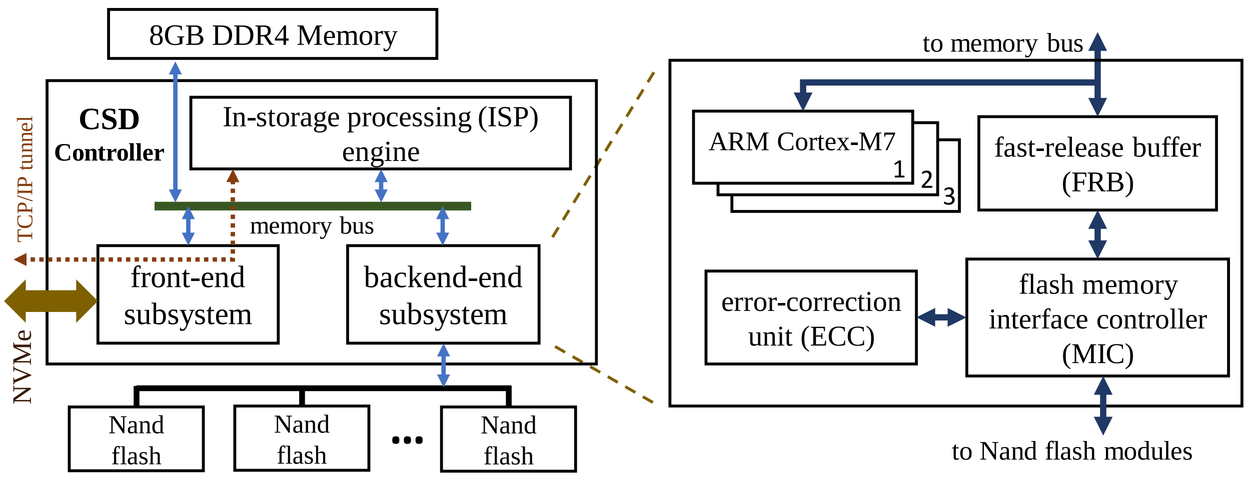


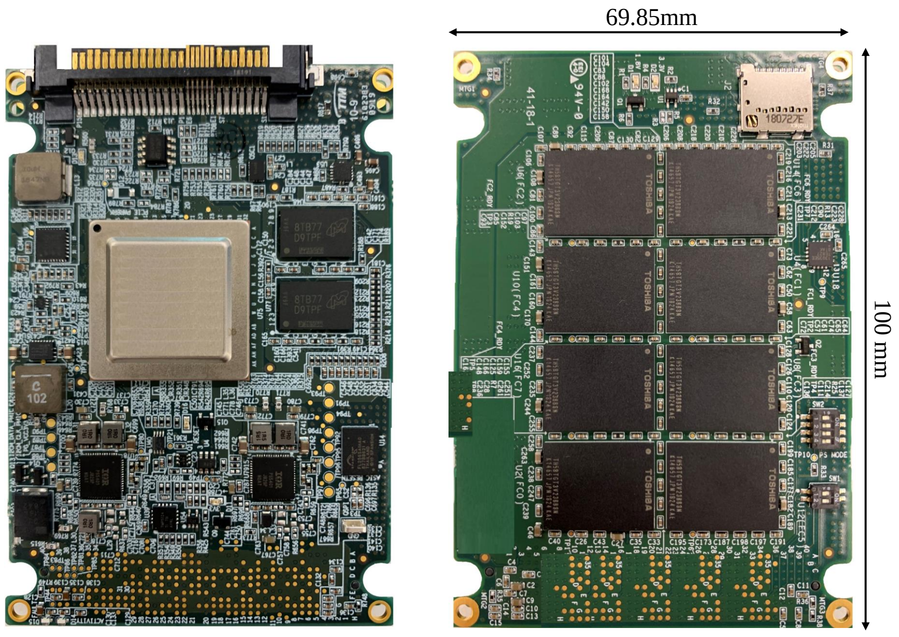
Figure 3.3: High-level architecture of Newport CSD.

off-chip computational storage solution. In addition, an Ethernet connection has been provided in CompStor to allow for a TCP/IP connection between the ISP engine and the host. Fig. 3.2 shows the prototype.

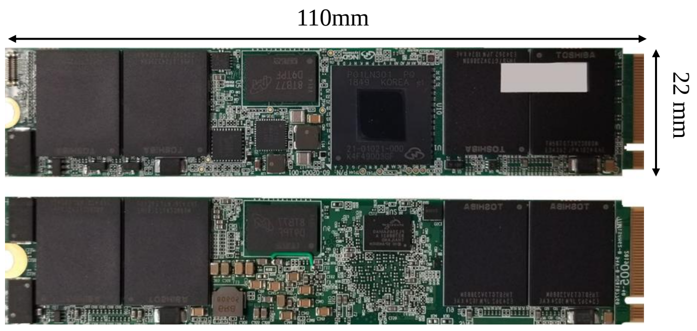
3.2.2 ISP Engine

The CSD controller is implemented by attaching an ISP engine via an FMC connector to the databus that also connects the conventional subsystems. 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+, which is an MPSoC chip containing an FPGA together with a quad-core 64-bit ARM Cortex-A53 processor.

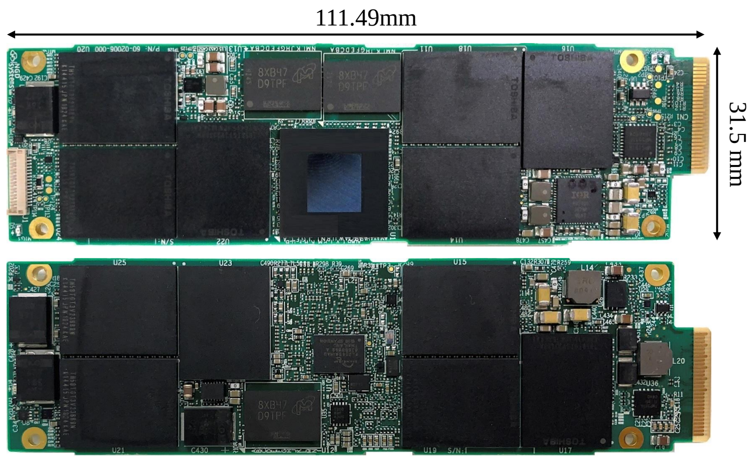
CompStor may be able to considerably improve the system performance and energy efficiency of I/O- and compute-intensive applications [24], but it is limited by the off-chip data transfer. As an ISP device, CompStor still needs to move data to/from the conventional subsystem. Although this data transfer is less expensive than the one via a complex NVMe interface, it is off-chip, which can incur higher latency and higher energy consumption compared to on-chip solutions.



(a) CSD in U.2 form factor



(b) CSD in M.2 form factor



(c) CSD in E1.S form factor

Figure 3.4: CSD prototypes in different form factors.

Table 3.1: Hardware specifications of Newport family CSDs.

Name	Form factor	Interface	DRAM	Capacity	Dimensions (mm)
Newport	U.2	NVMe over Gen3 PCIe x4	8GB	Up to 32TB	69.85 x 100 x 15
Laguna	M.2	NVMe over Gen3 PCIe x4	4GB	Up to 8TB	22.15 x 110 x 3.88
Solana	E1.S	NVMe over Gen3 PCIe x4	4GB	Up to 12TB	31.5 x 111 x 5.9

3.3 Newport's Hardware Architecture

To address the shortcomings in CompStor, we developed a more integrated, single-chip architecture for CSD controllers. We first prototyped it on an FPGA platform called Catalina [34]; however, due to inherent FPGA limitations, it became clear that FPGA-based CSD controllers cannot reach the maximum potential of in-storage processing capabilities. Therefore, the next family of single-chip ASIC-based CSD solutions, called Newport, was introduced. The Newport ASIC was designed by NGD Systems in a 14 nm CMOS FinFET process. The EDA tools, RTL design, and layout methodologies are mainstream for this process node. The ARM processors, PCIe, DDR, embedded SRAMs, and chip I/Os are IPs licensed by third-party IP providers. The rest of the logic was coded in Verilog and synthesized with tools from Cadence Design Systems.

Fig. 3.3 shows the high-level architecture of the CSD. The proposed CSD is composed of three main components, namely front-end, back-end, and ISP subsystems. The front-end (FE) and back-end (BE) subsystems are similar to the subsystems in the off-the-shelf flash-based storage devices, i.e., SSDs. Table 3.1 summarizes the specifications of our three prototypes while Fig. 3.4 shows the actual prototypes.

To manage the NAND flash modules, the BE subsystem contains three ARM Cortex-M7 processing cores together with a fast-released buffer (FRB), an error-correction unit (ECC), and a memory controller interface unit (MIC). The ARM Cortex-M7 cores run the flash translation layer (FTL) processes, including logical and physical address translation, garbage collection, and wear-leveling. The ECC unit is responsible for handling those errors in the flash memory modules. The FRB transfers data between the MIC and other components, while MIC issues low-level I/O commands to NAND flash modules. These modules are organized in 16 channels, so 16 IO transfers can be performed simultaneously.

3.3.1 Front-end

The FE is responsible for communicating with the host via the NVMe protocol. It receives the I/O commands from the hosts, interprets them, checks the integrity of the commands, and populates the internal registers to notify the BE that a new I/O command has been received. The FE also packetizes and depacketizes the data transferred to and from the host and the CSD. This subsystem consists of a single-core ARM Cortex-M7 processor and an ASIC-based NVMe/PCIe interface.

3.3.2 ISP Engine

Besides the FE and BE subsystems, there is also an ISP engine inside Newport, composed of a quad-core ARM Cortex-A53 processor running at 1 GHz and a software stack that provides a seamless environment for running user applications. The ISP engine has access to the shared 8 GB memory. A full-fledged Linux OS has been ported to the ISP engine, so the ISP engine supports a vast spectrum of programming languages and models. The ISP engine has a low-latency, power-efficient direct link to the BE subsystem to read and write the flash memory. In other words, the data transferred to the ISP engine bypasses the whole FE subsystem and the power-hungry NVMe/PCIe interface.

3.4 Newport Software Architecture

The ISP's dedicated embedded processors require different software layers to provide the underlying environment for executing various types of applications. By deploying a conventional embedded

Linux-based operating system, we enable a compatible environment to support a wide spectrum of applications, programming languages, and command lines without modifications. Considering the unique architecture of our CSD, the embedded Linux is extended with our own custom features, including device drivers, file systems, and communication paths.

3.4.1 Customized Block Device Driver

We developed a *customized block device driver* (CBDD) to enable access to the storage units that are optimized to the specific on-chip communication links and protocols, which are different from common protocols between processing units and storage devices such as PCIe and SATA. The CBDD uses a command-based mechanism to communicate with the back-end subsystem to initiate data transfers and receive their completions. Using the scatter-gather mechanism, the back-end subsystem handles data transfers through the DDR addresses exposed by the ISP's operating system.

3.4.2 Shared File System between Host and ISP

One unique feature of our CSD is that the CBDD supports file system access by the ISP applications and the host. The ability to mount partitions inside the ISP engine and access files through file systems makes it not only easier to port applications but also more efficient to access the data. Moreover, there is a software layer for file system synchronization between the host and the Newport ISP engine's operating systems. These two operating systems can access the data stored in the flash memory array at the file system level and concurrently mount the same storage media, which can be problematic without a synchronization mechanism [58]. We implemented the Oracle cluster filesystem 2nd version (OCFS2) [59] between the host and the CSD. Using the OCFS2, both the host and the ISP engine can issue flash I/O commands and mount the shared flash memory natively and simultaneously.

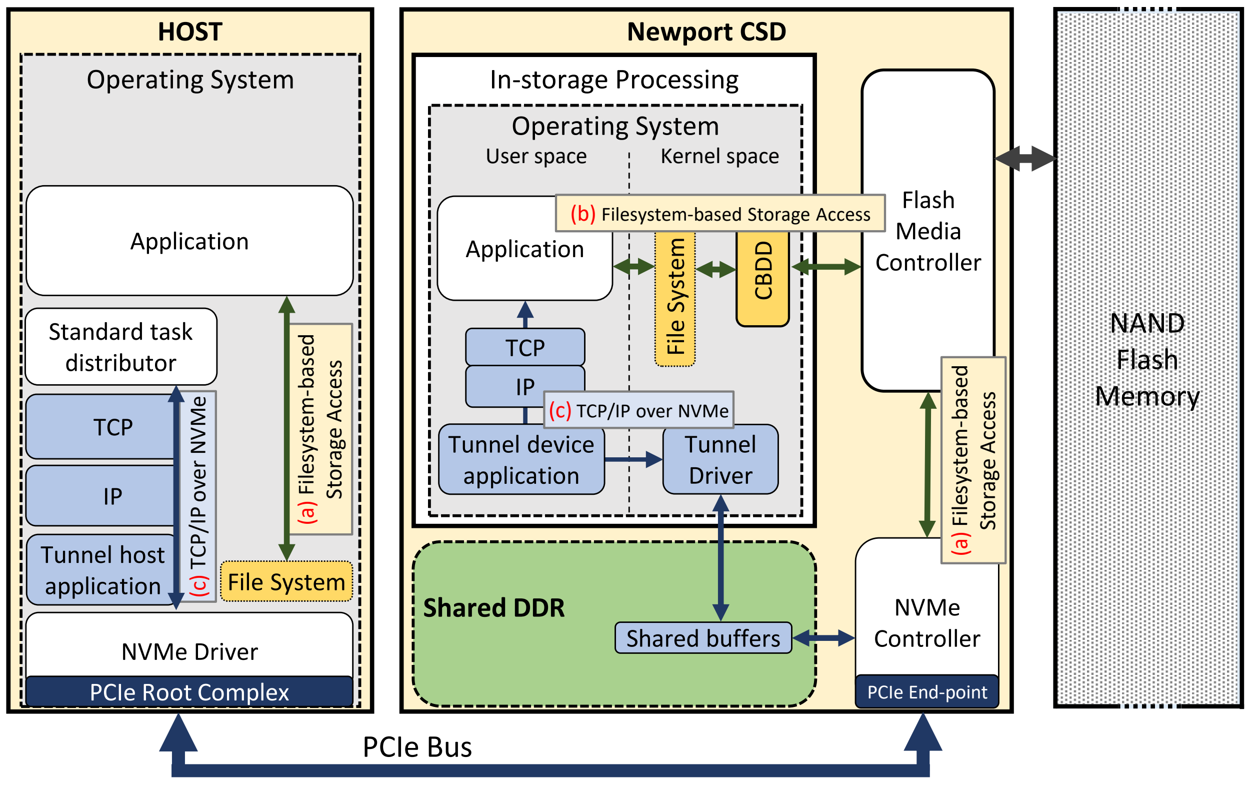


Figure 3.5: Newport software stack provides three different communication paths: (a) The conventional data path through the NVMe driver to the host, (b) The path through on-chip connection to the ISP subsystem with file-system abstraction, and (c) The TCP/IP tunneling over the PCIe/NVMe.

3.4.3 Communication paths

Fig. 3.5 depicts the different layers of the Newport software stack. It supports three communication paths over the different interfaces on our CSD: flash-to-host, flash-to-ISP, and TCP/IP tunneling.

The conventional data path (shown as path "a") is the flash-to-host interface that goes through NVMe protocol over PCIe to allow the host to access the data stored in the NAND flash memory. To do so, our NVMe controller is capable of handling NVMe commands by responding to them appropriately and managing data movement through communicating with our flash media controller.

Path "b" is the flash-to-ISP interface, which provides file-system-based data access to the ISP engine through an on-chip connection. This file-system-based data-access interface is implemented by CBDD through communication with our flash media controller. In fact, the flash media controller handles requests from both the ISP engine and the host.

Path “c” is for the TCP/IP tunneling over the PCIe/NVMe. From the user’s point of view, it is crucial to have a standard communication link between the host and the ISP engine. The user on the host side can use this communication link to initiate the execution of the application inside the ISP engine and monitor the outcome of the executions. To provide such a standard link, a TCP/IP tunnel through the NVMe/PCIe is supplied. In other words, host-side applications can communicate with the applications running inside the CSD via a TCP/IP link. This link is also essential for distributed-processing applications, where processes running on multiple nodes require a TCP/IP link to communicate [60].

In addition to enabling communication between the host (and the wide-area network) and the ISP system, this tunneling feature eliminates the need for unwieldy network setup. That is, many cables and switches would otherwise be required to connect the many tightly assembled storage units, which would be impractical to maintain and scale. The proposed TCP/IP tunnel uses two shared buffers on the onboard DDR to provide the data communication. Two user-level applications, one on the host and one on the ISP running in the background, are responsible for managing NVMe-based data movement and encapsulating/decapsulating the TCP/IP data through NVMe packets.

3.5 Summary

This chapter provides an overview of the fundamental architecture of Solid State Drives (SSDs), which comprises three main components: the NAND flash, the controller, and the front end. These components are responsible for data storage, data processing and management, and the interface between the storage device and the host system, respectively. Additionally, SSDs conform to certain design standards and environmental factors, referred to as form factors.

Computational Storage Drives (CSDs) inherit the architecture of conventional SSDs, with the inclusion of extra processing subsystems along the controller unit. However, our initial prototype of CSD, named CompStor, suffered from the drawback of data transfer between the controller unit and the off-chip ISP engine. In the subsequent generation of CSD controllers, named Newport, the units have been integrated to develop an ASIC-based controller, featuring an internal ISP engine that includes one quad-core ARM Cortex-A53 processors. The integrated ISP engine enjoys seamless access to the NAND flash data through the shared DRAM with the controller. To ensure data consistency between the host and the ISP, a customized block device driver has been designed. Furthermore, the software stack enables the execution of an operating system on the ISP, as well as direct high-speed communication between the host and the operating system running within the CSD.

The subsequent chapter will delve into the applications that can derive substantial benefits from the CSD design.

Chapter 4

Distributed Training on Computational Storage Drives

Distributed training of deep-learning models can be done efficiently on CSDs while preserving data privacy. This section first reviews the types of parallelization applicable to deep neural networks so they can run in a distributed environment with heterogeneous or homogeneous nodes. We propose a framework named Stannis for distributing training workload onto a network of CSDs and a method called Hypertune for adapting the hyperparameters for quicker convergence[61, 62, 63].

4.1 Model and Data Parallelization

Training deep neural networks (DNNs) is a computationally intensive task that requires large amounts of data and computing resources. One practical approach to reducing the training time is to parallelize the training task onto multiple processors. Two common methods for parallelizing such tasks are model-parallel and data-parallel training.

Model-parallel training divides the DNN into multiple sub-networks, each of which is trained on

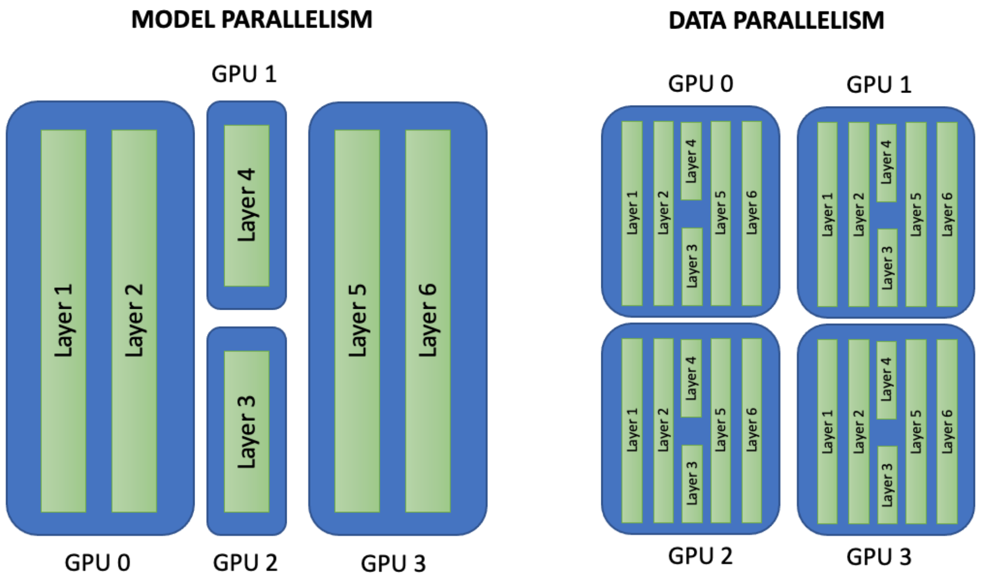


Figure 4.1: Model parallel vs. data parallel [1].

a separate processor. This approach is well-suited for DNNs with large numbers of parameters, as it can significantly reduce the memory requirements of training. However, it can be difficult to implement model-parallel training for DNNs with complex topologies.

Data-parallel training divides the training data into multiple subsets, each of which is processed by a separate processor. This approach is simple to implement and can be used to train DNNs with any topology. However, it can require a large amount of communication between processors, which can slow down the training process. Fig. 4.1 shows the architecture of both approaches.

Several well-known AI frameworks, such as Tensorflow [64], Pytorch [65], Theano [66], and Horovod [67], have implemented distributed training of DNNs. These frameworks provide a variety of features that make it easier to develop and deploy distributed training applications. Despite the availability of these frameworks, distributed training of DNNs can still be challenging. One of the main challenges is managing the communication between processors. This can be a complex task, especially when the processors are distributed across a network. Another challenge is ensuring that all processors have access to the same data. This can be difficult to achieve, especially when the data is stored on a distributed file system.

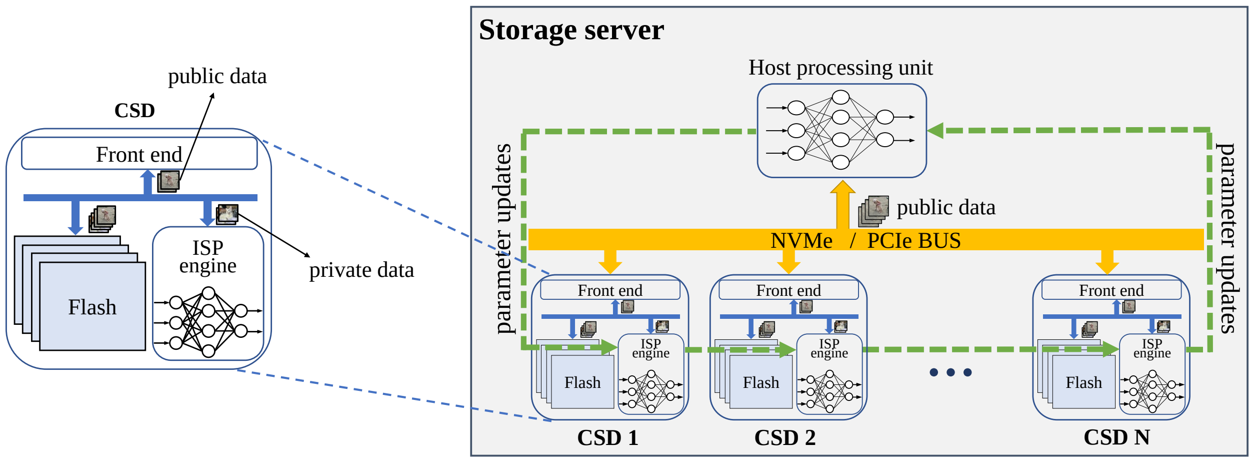


Figure 4.2: Distribution of training workload onto CSDs and host using Stannis.

Despite these challenges, distributed training of DNNs is a powerful tool that can be used to train DNNs more quickly and efficiently. As the size and complexity of DNNs continue to grow, distributed training will become increasingly important.

4.2 Stannis: Framework for Training NN in Storage

In response to the requirements of our proposed methodology, we have developed a novel framework called Stannis, which is based on Horovod. Stannis, which stands for *System for TrAining of Neural Networks In Storage*, is designed to enable the model-parallel distribution of neural network training on both homogeneous and heterogeneous computing systems. Although Horovod can achieve significant speedup on homogeneous systems, it faces challenges on heterogeneous systems, where the slowest processor becomes the bottleneck due to the need for synchronization during training. This implies that faster processors are constrained by slower processors in the system.

To overcome Horovod's inability to work efficiently on heterogeneous systems, Stannis employs a workload-scaling strategy that considers transmission delay and data privacy concerns. Specifically, Stannis addresses the challenges associated with data transmission by adjusting the workload distribution based on the delay time incurred during data transmission. Additionally, Stannis incorporates a data privacy mechanism to protect the privacy of sensitive data during transmission. By using these strategies, Stannis enables efficient neural network training on both homogeneous and heterogeneous computing systems, improving the scalability and speed of the training process.

4.2.1 Time Equalization by Benchmarking

To ensure that all processing engines operate without stalling, Stannis adopts a strategy that seeks to equalize processing time among all nodes by assigning different batch sizes to each processing engine. Specifically, slower processors receive smaller batch sizes to allow them to complete the training of each epoch within the same elapsed time as the more powerful processors. The pseudocode for Stannis is presented in Algorithm 1.

Stannis begins by conducting a series of benchmark tests on all processing engines to evaluate their processing speeds and determine the optimal batch size for each processor. Based on the benchmark results, the best batch size is selected for the slowest engine a.k.a. CSD, as it has more accumulated total processing power and a greater impact on the overall system's performance. Furthermore, slower engines are more sensitive to changes in batch size than the more powerful processors. Therefore, the optimal batch size for the slowest processor is determined first, followed by the selection of batch sizes for the more powerful processors. This approach ensures that each processing engine operates efficiently without becoming a bottleneck in the training process, thereby improving the overall performance of the system.

4.2.2 Batch Size Adaptation

Once the optimal batch size has been identified for the slowest processor, Stannis proceeds to determining the best batch size for the other processing engines. This is achieved by first calculating the time it takes for one batch to complete on the slowest processor. Then, the batch size for the other processing engines is increased by a fraction $\left(\frac{1}{C}\right)$ of the time difference between the slowest processor and each of the other processing engines, until comparable elapsed times are achieved. Here, C is a constant that determines the magnitude of the batch size adjustments, with larger values of C resulting in more fine-grained updates to the batch size. Additionally, Stannis accounts for the synchronization overhead that occurs during the training process and allocates a margin of $\left(\frac{time}{E}\right)$ to the final time, where E is a factor that is determined by observing the slowdown pattern that occurs when new processing nodes are added to the system. The value of E is set to allow a fixed 20% margin in the results. By incorporating these adjustments, Stannis ensures that the training process operates efficiently on all processing engines while minimizing stalling and synchronization issues.

4.2.3 Data Localization

In addition to optimizing the batch size and considering synchronization overhead, Stannis also takes into account the security of the data being processed. Specifically, Stannis assigns private data to the local ISP engine, while public data is shared between the ISP engine and the host processor. This approach minimizes the transmission of private data over the network and to the host, thereby increasing the level of data protection.

The overall architecture of running Stannis on a data server is depicted in Fig. 4.2. By incorporating these security measures, Stannis ensures that data privacy is maintained while achieving efficient and effective neural network training on a distributed system.

4.3 HyperTune: Adaptive Scheduler for Time Equalization

HyperTune is a run-time mechanism incorporated into Stannis to ensure that all processing nodes experience minimum stalling during distributed training[1] of a neural network. While Stannis

[1]sometimes also called *federated learning* (FL), though FL is not necessarily synchronized.

Algorithm 1: Stannis's Tuning algorithm

```
Input: $IP_{\text{CSD}}, IP_{\text{host}}, C$
Output: $(BS_{\text{host}}, BS_{\text{CSD}})$
Function Tune($IP_{CSD}, IP_{host}, C$):
    for batch sizes in list of BS do
        run benchmark on CSD;
        update $BS_{CSD}$ to the best one;
        update $time_{CSD}$;
    end
    Let $E$ = margin scale ;
    while $(time_{host} - time_{CSD}) < (time_{CSD}/E)$ do
        $BS_{\text{host}}$ += $BS_{\text{host}} \times (time_{\text{CSD}} - time_{\text{host}})/C$ ;
        run benchmark on host ;
        get the $time_{\text{host}}$ ;
    end
    return $(BS_{\text{CSD}}, BS_{\text{host}})$;
End Function
```

attempts to allocate workload proportionally to the estimated performance of each CSD, this alone may not guarantee equal execution time due to several factors.

For instance, processors may be time-shared with other tasks, which can reduce the available processing time for the training session. Furthermore, although the batch size can be adjusted based on feedback, some hysteresis must be built-in to ensure stable operation. To compensate for workload interruption in nodes, Stannis uses the HyperTune function, which is specifically designed to dynamically adjust the batch size and prevent stalling by redistributing workload among processing nodes. By incorporating HyperTune, Stannis can achieve optimal performance and minimize training time on a distributed system comprised of heterogeneous nodes.

4.3.1 Tuning Mechanisms

HyperTune is a dynamic runtime mechanism that aims to reduce the stalling of processing nodes during the model-parallel distribution of neural-network training. The mechanism achieves this by rescheduling the operation assigned to each node based on the availability of processing cycles.

This is accomplished by continually measuring the local processing speed and available processing power on each node and updating the batch size list accordingly. In cases where a node is busy, the batch size assigned to that node is decreased, while the batch size assigned to other nodes is increased to maintain a balance of processing power across all nodes.

When batch sizes are changed, Stannis reassigns the dataset based on the new batch size to prevent rank stall during the training session. To achieve this, the dataset's indexes and lengths are passed to each node using the `MPI_scatter()` function. To monitor the performance of the nodes during the training session, a monitoring session is implemented after each step within the epoch. Speed measurements from all nodes are gathered on an arbitrary node using `MPI_gather()` and passed to a decision-making function.

To make better decisions, the decision-making function receives information about the speed change and the percentage progress of the current epoch. This information is then converted to a declining index based on the weighted sum in Equation (4.1).

$$index_i = 0.7 \times \frac{SP - SP_i}{SP} + 0.3 \times \left(\frac{N_{\text{step}} - step_i}{N_{\text{step}}} \right) \tag{4.1}$$

In the equation, *SP* represents the normal speed obtained from the `batchsize_to_speed()` function. SP_i and $step_i$ represent the current speed and step, respectively, while N is the number of steps per epoch. By constantly adjusting the batch sizes based on the availability of processing cycles and monitoring the performance of the nodes during the training session, HyperTune ensures that stalling by any processing node is minimized, thus improving the efficiency of model-parallel distribution of neural-network training.

4.3.2 Hysteresis in Adaptation

To ensure the stable operation of the system, a hysteresis algorithm has been implemented in HyperTune to prevent chattering caused by glitches or mis-measurements in the processing speed. In this algorithm, a threshold of 25% has been set for the declining index, beyond which the current step is flagged as under-utilized, and this information is stored in a separate array. If five consecutive under-utilization flaggings occur, the current epoch is terminated, and the `batchsize_controller()` function is triggered to determine a new batch size.

The implementation of the hysteresis algorithm serves as a crucial buffer against abrupt and transient changes in processing speed that could otherwise lead to unstable system operation. Specifically, the algorithm sets a threshold for the declining index and requires a certain number of consecutive under-utilization flaggings before triggering the `batchsize_controller()` function. This approach enables the system to respond to genuine changes in processing speed while ignoring minor fluctuations that could lead to unnecessary batch size adjustments.

By requiring a certain number of consecutive under-utilization flaggings, the algorithm effectively reduces the impact of chattering, which is the rapid switching between two states due to noise or other sources of disturbances. This is particularly important in the context of batch size adjustments, as frequent and abrupt changes to the batch sizes can destabilize the training process, leading to poor performance and longer convergence times.

Overall, the hysteresis algorithm contributes to the stability and efficiency of the system by providing a robust and reliable mechanism for adjusting batch sizes based on changes in processing speed.

Initially, we attempted to utilize the inverse of the `batchsize_to_speed()` function acquired from the tuning phase to determine the new batch size based on the current processing speed of the interrupted node. However, after conducting evaluations, we discovered non-negligible errors that had the potential to deteriorate system performance.

As a second approach, we opted to use a weighted averaging method from the closest values to the present speed. The new batch size calculation is derived from Equation (4.2):

$$BS_i = BS_n \times \frac{SP_i - SP_n}{SP_{n+1} - SP_n} + BS_{n+1} \times \frac{SP_{n+1} - SP_i}{SP_{n+1} - SP_n} \tag{4.2}$$

where BS_i is the new optimal batch size, SP_i is the current speed between SP_n and SP_{n+1}, and BS_n and BS_{n+1} are the batch sizes corresponding to SP_n and SP_{n+1}.

Another approach, which provides nearly the same results without imposing additional processing costs, is to monitor the CPU usage of the training session on each node. We implemented a sliding window to keep track of the CPU usage for the last ten steps. The new batch size is proportional to the average of the last five steps (the second half of the sliding window) with the declined CPU usage and the normal CPU usage (stored in the first half of the sliding window). The window size should be sufficiently large to disregard possible glitches but not excessively large, so it can detect the utilization decline pattern.

4.3.3 Hyperparameter Tuning

It should be noted that various parameters, including the size of the sliding window and the margin for speed decline detection, are of an experimental nature and are subject to adjustments based on the desired level of precision. An advantage of employing CPU utilization as a measure of performance is that the system can increase the batch size when additional processing cycles are at hand. This is particularly beneficial, as the training session can reclaim the resources that were previously occupied, thereby boosting overall performance. In contrast, the speed-monitoring approach is unable to detect the availability of extra processing cycles.

To counteract the overall performance decline caused by node interruption, Stannis rapidly monitors the performance and updates the hyperparameters, thereby assigning a larger portion of the processing load to the free nodes. However, this approach raises a concern that by terminating the epoch prematurely, the network might miss some of the data in the training process, and multiple occurrences of such terminations might result in complete data loss for that portion of the dataset. To address this concern, the shuffle function is utilized when creating the input batch of the data. This function ensures that the data that is assigned to the batch is randomized, which statistically makes it very likely that all input data will undergo training after a sufficient number of epochs. By using the shuffle function, the network can overcome the challenge of epoch termination and reduce the likelihood of data loss.

Another potential issue that arises when dynamically adjusting the batch size is the impact on the convergence of the training process. To alleviate this concern, it is important to restrict the range of batch-size changes to ensure that the optimization process can still converge effectively. In addition to batch size, careful design of the model's architecture and dynamic selection of other hyperparameters, such as the learning rate, can also contribute to better convergence rates and higher achieved accuracy.

For instance, in a recent study on spatio-temporal forecasting, [68] found that dynamic selection of hyperparameters, including batch size and learning rate, could significantly improve the performance of deep-learning models. Specifically, they proposed a decision-making function to adjust the learning rate in real time based on the training progress and the model's accuracy. This approach could lead to more stable convergence and faster optimization of the model's parameters. Future work could extend this approach to also consider the dynamic adjustment of batch size based on the available processing resources and the training progress.

4.4 Training of Large Language Models (LLM) on CSD

Large Language Models (LLMs) represent a groundbreaking advancement in artificial intelligence (AI) and natural language processing (NLP), transforming the landscape of human-machine interaction and language-related tasks. These models, such as OpenAI's GPT series and Google's BERT, are characterized by their massive scale, containing billions or even trillions of parameters [69]. At their core, LLMs are built upon transformer architectures [70], which have revolutionized NLP by enabling models to handle long-range dependencies in text efficiently. The transformer architecture employs self-attention mechanisms that allow the model to weigh the importance of different words in a sentence, capturing contextual information and relationships between words [70]. This architectural innovation has been instrumental in enabling LLMs to learn intricate linguistic patterns and nuances from vast amounts of text data.

LLMs undergo extensive training on massive datasets comprising billions or even trillions of words to learn the structure, semantics, and patterns of human language [71]. This unsupervised learning process equips LLMs with the ability to generate coherent and contextually relevant text, making them invaluable tools for a wide range of NLP tasks, including machine translation, text summarization, sentiment analysis, and chatbots [69]. The applications of LLMs span across diverse domains, showcasing their versatility and potential impact on various industries. In healthcare, LLMs can assist in analyzing medical literature and patient data, aiding in diagnosis and treatment planning [72]. In finance, LLMs can be utilized for sentiment analysis to gauge market trends and predict financial outcomes. Moreover, they power chatbots and virtual assistants, enhancing customer service and user experience, while also serving as content generation tools in entertainment and media [73].

Despite their impressive capabilities, LLMs face significant challenges, especially when it comes to computational resource limitations, memory constraints, communication overhead, and integration complexity. These challenges are particularly evident when considering the use of Computational Storage Drives (CSDs) for LLM training. One major challenge of using CSDs for LLM training is their limited computational resources compared to dedicated compute nodes with high-performance CPUs or GPUs [74]. LLM training involves extensive matrix multiplications and complex operations that may overwhelm CSDs, leading to slower training and suboptimal performance. Additionally, the parallelism crucial for efficient LLM training might be constrained by the limited processing power of individual CSDs, hindering scalability and distributed computing capabilities.

Beyond limited computational power, CSDs often have restricted onboard memory compared to traditional compute nodes. LLMs with millions or billions of parameters require significant memory to store model parameters and training data. The limited memory capacity of CSDs might hinder the efficient storage and manipulation of LLM components, creating performance bottlenecks and potentially compromising training quality. Furthermore, communication between CSDs to exchange data and synchronize computations can exacerbate memory constraints, further impacting training efficiency.

In distributed training scenarios using multiple CSDs, communication overhead between storage devices becomes a significant bottleneck [75]. Coordinating data exchange and synchronization among distributed CSDs introduces latency and increases training time, diminishing the potential benefits of parallel processing. The need to transfer large volumes of data during training, especially when parameters or data are distributed across multiple devices, exacerbates communication overhead.

Integrating LLM training algorithms with CSD architectures introduces additional complexity in software and hardware design. Ensuring compatibility and optimizing performance across diverse storage and computational platforms requires specialized expertise and incurs development overhead. Additionally, adapting existing LLM training frameworks and algorithms to leverage CSDs effectively may necessitate significant modifications, potentially disrupting established workflows and introducing new challenges.

While Computational Storage Drives offer intriguing possibilities for accelerating various computational tasks, including neural network training, their suitability for Large Language Model training remains limited [76]. The combination of limited computational resources, memory constraints, communication overhead, and integration complexity make CSDs less effective for efficiently training LLMs compared to dedicated compute infrastructure. As LLMs continue to evolve and grow in complexity, alternative approaches leveraging specialized hardware architectures and dedicated computational resources are likely to remain the preferred choice for efficient and scalable training.

4.5 Summary

This chapter presents our software solution, Stannis, for the efficient distribution of deep-learning models across multiple processing nodes such as Computational Storage Drives (CSDs). Stannis aims to optimize the performance of distributed deep learning by dynamically assigning workloads to CSDs based on their processing capabilities. In particular, Stannis strives to balance the processing time of each step to minimize synchronization wait time and assigns private data to local processors to preserve data privacy.

Moreover, Stannis employs a performance-monitoring function called HyperTune to mitigate the impact of any external interrupt on the training process. HyperTune dynamically monitors the system's performance during training and reallocates workloads among the processing nodes if necessary, ensuring that the training process continues with minimal impact. Through HyperTune, Stannis can achieve efficient and robust distributed training of deep-learning models on CSDs. However, despite Stannis' capabilities in distributing various models, the limitations in computational resources, memory, communication overhead, and integration complexity make training large language models (LLMs) on CSDs less effective compared to dedicated computing infrastructure. As LLM complexity grows, specialized hardware and dedicated computational resources will likely remain the preferred choice for efficient and scalable training.

The next chapter discusses other potential applications of CSDs to highlight the benefits of using CSDs in these scenarios. Specifically, we explore how CSDs can be utilized to reduce the data movement and communication overheads involved in communication-intensive applications.

Chapter 5

Applications

This chapter presents the various applications utilized to evaluate the potential of the CSD framework in data analytics. Apart from the distributed training of deep learning models, as discussed in the previous chapter, we conducted benchmarking tests on additional applications, such as distributed inferencing, database lookup, and search and analytics engines.

5.1 Distributed Inferencing for Natural Language Processing

Distributed inferencing refers to the process of distributing the computational load for inference tasks among multiple nodes in a network. Unlike distributed training (Chapter 4), which involves updating the parameters of the model based on feedback, distributed inferencing is a simpler process that does not require closed-loop feedback. It can be performed using either model-parallel or data-parallel schemes, both of which are designed to improve overall performance. On the other hand, Natural Language Processing (NLP) is a class of applications that lends itself well to parallelization, given the high workload demands of NLP services such as speech-to-text converters, text auto-correction, content recommendation, and filtering. These services are widely used in various domains, from automated call centers and search engines to voice assistants such as Apple Siri and Amazon Alexa. Scalability is a crucial requirement for such services to handle the ever-increasing number of queries in a timely manner. For instance, Google's search engine processes an average of 63,000 search queries per second, and Netflix uses 1,300 recommendation clusters based on user preferences to filter over 3,000 titles at a time[77, 78]. The efficient handling of these queries necessitates advanced algorithms and powerful hardware. The importance of NLP services is underscored by the increasing amount of data available in the form of text and speech. Such data is generated by social media, web content, and other digital sources. The exponential growth of data coupled with the need for real-time processing creates a pressing demand for scalable and efficient NLP solutions. Parallelization techniques can help meet these requirements by dividing the computational load among multiple computing nodes, thus speeding up the processing time. Additionally, the availability of distributed hardware architectures such as clusters and Cloud Computing provides an opportunity to leverage distributed computing to meet the high demands of NLP services.

As such, the use of parallelization techniques to enable efficient and scalable NLP services is an active area of research. Researchers are developing parallel algorithms and models to handle large-scale NLP tasks, such as language modeling, machine translation, and sentiment analysis. The use of hardware accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) is also gaining popularity as they can significantly speed up NLP tasks. Overall, parallelization techniques hold the promise of enabling the efficient and scalable processing of the vast amounts of natural language data generated by various digital sources.[77, 78].

NLP applications often require large models of up to tens of gigabytes, resulting in high I/O intensity. As a consequence, there is a need for frequent loading and swapping between the main memory (DRAM) and disk, leading to extended latency and increased energy consumption during data transfer between storage and the host. The size of the model is largely dominated by embedding tables or layers that contain several parameters that describe each entry. In most cases, determining the final answer to a query involves dispatching the processing on the neighboring node closest to the input data. NLP applications typically require preprocessing, which includes *tokenization* (breaking down the text into smaller semantic units), *word tagging* (assigning parts of speech to words), *stemming* or *lemmatization* (standardizing words by reducing them to their root forms), and *stop-word removals* (filtering out common words that add little or no unique information) such as prepositions and articles (“at”, “to”, “a”, “the”).

In light of these factors, we posit that in-storage processing can significantly enhance the performance of NLP and many other applications with similar characteristics by employing model-parallel and data-parallel schemes. In the model-parallel approach, in-storage processing is used to partially process the data residing on the flash before sending it to other nodes, such as the host CPU or GPU, for further processing. This technique is particularly useful for neural network applications that require preprocessing of the raw input data. Additionally, it improves the performance of neural networks that saturate a system’s DRAM with large embedding tables, requiring high I/O intensity but relatively low computation. This model has been successfully implemented in Facebook’s deep learning recommendation model (DLRM) project [79], where enormous embedding tables exceeding tens or hundreds of gigabytes are processed on specific nodes, and only the output is sent to the upper nodes for the remaining processing.

In NLP application use cases, CSDs can be used both in model-parallel and data-parallel scheme. In the model-parallel method, in-storage processing can be leveraged by CSD to partially process data that resides on the flash before transmitting it to other nodes, such as the host CPU or GPU, for further processing. This technique is especially useful for neural network applications that require preprocessing of the raw input data. Furthermore, it improves the performance of neural networks that saturate the system’s DRAM with large embedding tables, which necessitate high I/O intensity but involve relatively small computation. [79, 80, 81]. In the data-parallel method, each processing node is responsible for running the entire model on its local input data before sending it back to the supervising node. For this application, we have deployed a data-parallel scheme to maximize the benefits of in-storage processing.

It should be noted that the use case of in-storage processing is not limited to NLP applications. ISP presents a universal solution for enhancing system performance across any application that may be executed in parallel. The next section will describe a general approach to run a spectrum of applications on CSDs.

5.2 Query Scheduler for Workload Distribution

In order to effectively distribute the processing tasks between the host system and the CSDs with minimal overhead, we have developed a scheduler that can distribute applications across multiple nodes. This scheduler is based on MPI and was implemented in the Python programming language. It has the capability to redirect requests or queries to available nodes. The pseudocode of the scheduler is presented in Algorithm 2 [82].

The scheduling process commences by executing the tuning algorithm, which is a reduced-scale version of the application, on the CSD. The objective of this algorithm is to determine the most appropriate batch size, in terms of query-processing speed. As the batch size increases, the latency of processing each query generally increases, while the overall processing speed increases until communication becomes the new limiting factor. Therefore, the optimal batch size is selected by finding the smallest batch size that yields a processing speed closest to the maximum. This is achieved by comparing the speedup for each new batch test, and the tuning process concludes when the difference is less than a predefined constant, referred to as *margin*. If the difference exceeds this threshold, the largest batch size is selected.

Subsequently, the same test is conducted on the host to identify its optimal batch size. The scheduler's critical parameters are the two optimal batch sizes for the host and the CSDs, as well as the batch ratio (BR), which is the ratio between the two batch sizes. The BR is determined based on the difference in processing performance in a heterogeneous system, and it is used in subsequent runs. Since the host's CPU (Xeon) is more powerful than that of the CSD (ARM A53), the BR is significantly large, typically ranging from 20 to 30, as observed in our experiments. Any BR other than the optimal BR results in the under-utilization of the system. Employing the BR reduces the scheduler's workload, decreases scheduling overhead, and enables larger chunks of data to be processed at a time, thereby enhancing the host's performance.

After determining the optimal batch sizes, each node is assigned one batch of queries to process, and upon completion, the node sends an acknowledgment (ack) to the scheduler, which also acts as a request for the next batch. The scheduler operates in a separate thread on the host, waking up every 0.2 seconds to check for a new completion message from the working nodes. By putting the scheduler to sleep, the thread frees the host processor, thus increasing the available processing capacity. The shared-disk file system employed in our setup is the Oracle Cluster File System 2 (OCFS2), which enables both the host and the ISP to access the same shared data stored on flash. Consequently, the scheduler transmits only data indexes or addresses to the ISP engine, thereby considerably reducing communication overhead and eliminating one of the primary bottlenecks in parallel systems. Additionally, this method enhances the read/write speed, as all nodes access the data at a much higher speed when communicating directly with the flash. In contrast, the data access speed over TCP/IP is typically in the range of 10s of MBps, whereas the host and ISP engine's data access speed is on the order of GBps[83].

5.3 Database Applications

In the subsequent section, we will discuss the advantageous aspects of employing CSD within the context of database applications. Specifically, we will utilize MongoDB as an illustrative example of a prominent database application, discussing both its merits and demerits, while highlighting the potential enhancements that can be achieved through the integration of CSD, thereby optimizing

Algorithm 2: Query scheduler for distributed inferencing

```
Function TuneBS():
    for bs in range of BATCHSIZE for CSD do
        run benchmark on CSD;
        if $speed_{current} - speed_{CSD} >$ MARGIN then
            update $BS_{\mathrm{CSD}}$ to bs;
            update $speed_{\mathrm{CSD}}$ to $speed_{\mathrm{current}}$;
        end
    end
    for bs in range of BATCHSIZE for Host do
        run benchmark on Host;
        if $speed_{current} - speed_{host} >$ MARGIN then
            update $BS_{\mathrm{host}}$ to bs;
            update $speed_{\mathrm{host}}$ to $speed_{\mathrm{current}}$;
        end
    end
    let $BR = BS_{\mathrm{Host}}/BS_{\mathrm{CSD}}$;
    save BR for future use;
    return $(BS_{\mathrm{CSD}}, BS_{\mathrm{host}})$;
End Function
Function Query scheduler($IP_{CSD}, IP_{host}, Dataset$):
    run TuneBS();
    run the main application;
    while dataset left do
        wake up every 0.2 seconds;
        if completion message from any node then
            send indexes for the next query batch to the node;
        end
    end
End Function
```

its overall performance.

5.3.1 MongoDB

MongoDB is a widely used document-oriented database management system that was introduced in 2009. It is regarded as a next-generation database system that belongs to the NoSQL category[84]. The system was developed to address the scalability issues that relational databases experience when handling large amounts of data, which is typical of modern applications. While relational databases are efficient at processing small amounts of data, they struggle with data-intensive applications such as big data, IoT, machine learning, social media, and multimedia.

MongoDB's approach to data storage involves the use of dynamic schemas for JSON-like documents, stored in the BSON format. The system is renowned for its flexibility, speed, power, and ease of use [85], making it an attractive option for many applications. High availability and scaling are two of the primary challenges that databases often face.

To address these challenges, MongoDB offers several features. For instance, it provides built-in support for horizontal scaling and automatic sharding, enabling the system to distribute data across multiple servers. Additionally, MongoDB employs a distributed architecture, which replicates data across multiple nodes to ensure high availability. The system supports automatic failover, where a primary node fails, and a secondary node automatically takes over to minimize downtime. MongoDB also offers features like data compression, indexing, and aggregation for faster processing of large data sets.

Despite its advantages, MongoDB also faces some challenges, such as data consistency, security, and reliability. As a document-oriented database, it lacks ACID compliance, which means that it may not always ensure data consistency. MongoDB also requires additional configurations to ensure security and reliability. Nevertheless, MongoDB remains a popular choice for many modern applications that require fast, scalable, and flexible data storage solutions.

5.3.2 Sharding for Scalability

Sharding is a well-established technique for scaling databases, which involves partitioning a database into multiple database chunks, known as shards, and storing them on separate storage drives. Database scaling can be achieved through two methods, namely vertical and horizontal scaling. The *vertical scaling* method involves upgrading the capabilities of individual servers, such as increasing computing power and memory. However, upgrading server components requires a significant amount of effort and is not always feasible for rapid scaling. In contrast, *horizontal scaling* distributes large data sets across multiple servers, providing a more practical and scalable approach to database scaling. The fundamental principle underlying horizontal scaling is parallel processing, which is key to achieving high levels of database scalability.

Horizontal scaling offers several benefits over vertical scaling, including improved fault tolerance, higher availability, and increased processing power. By distributing data across multiple servers, horizontal scaling enhances the ability of databases to handle massive volumes of data while improving the overall performance of the system. Additionally, horizontal scaling can be achieved using commodity hardware, which is less expensive than specialized hardware required for vertical scaling.

However, sharding can also present some challenges, such as managing shard distribution, maintaining data consistency, and ensuring fault tolerance. These challenges can be addressed through the careful design and implementation of sharding strategies, such as using data replication, load balancing, and consistent hashing algorithms.

Overall, horizontal scaling through sharding is an effective technique for scaling databases and handling massive data sets. The technique provides a practical and cost-effective approach to database scalability, with the potential to improve system performance, reliability, and fault tolerance.

MongoDB offers the capability to shard data across multiple storage drives, which enables the system to leverage the efficiency and performance of parallel dynamics. MongoDB distributes the shards across multiple servers. By doing so, MongoDB can take advantage of parallel processing, which is crucial for enhancing the scalability of databases. The number of shards per server corresponds to the number of available storage drives. For instance, a 1U server that has 32 8-TB CSDs can create up to 32 shards with a total of 128 processing cores. This provides an opportunity to generate a node from each CSD while offering a colossal integrated capacity of 256 TB.

As illustrated in Fig. 5.1, a sharding scenario can be established for a host and nine NGD CSDs. This example demonstrates how the sharding technique can be applied to improve the performance and scalability of a MongoDB-based system. The host and CSDs communicate through a shared-disk file system, allowing the nodes to access the same data stored on the flash memory. By distributing the data across multiple drives, the sharding process significantly enhances the processing capacity and efficiency of the system. Moreover, the parallel processing capabilities provided by sharding enable the database to handle larger data sets and support a more extensive range of data-intensive applications.

5.3.3 Replication for Availability

In MongoDB, replica sets play a vital role in ensuring data redundancy and high availability. High availability refers to the capability of a system to remain operational and provide uninterrupted service in the face of various failures. To achieve high availability, MongoDB provides multiple copies of data on different database servers, known as replica sets.

In the event of a primary-node failure, eligible secondary nodes participate in an election process to select a new primary node. This process ensures that the system can operate continuously without

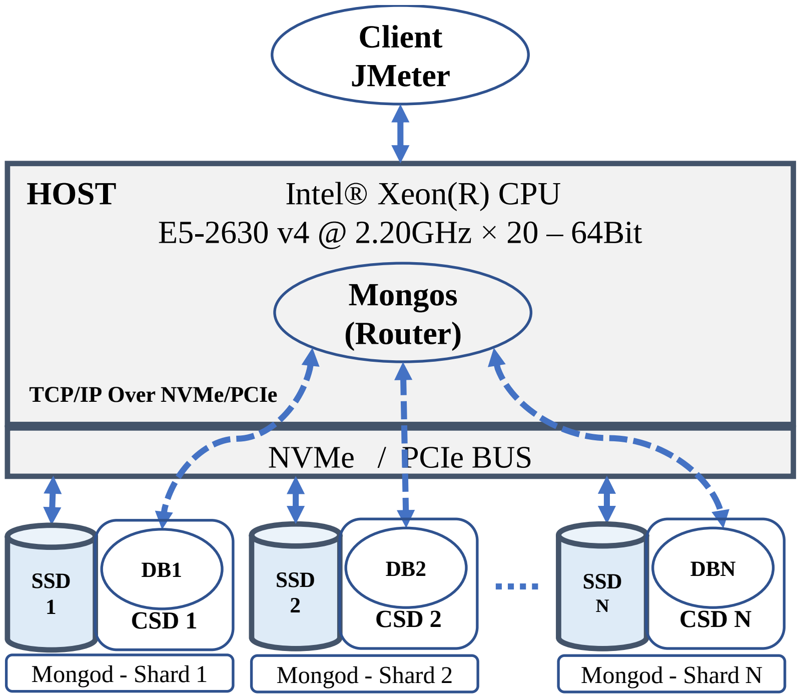


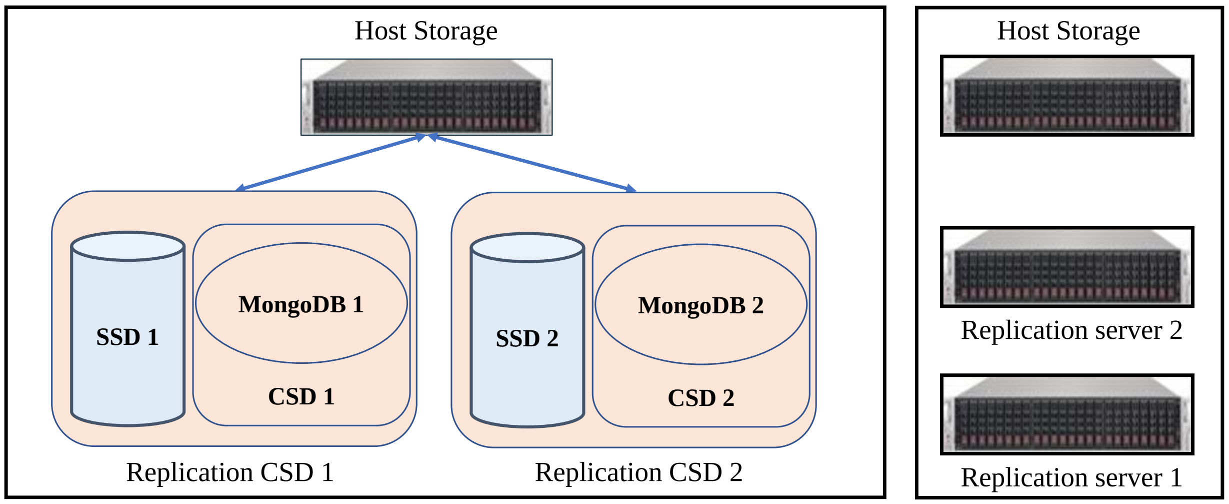
Figure 5.1: A sharding scenario for a host and *N* CSDs.

any downtime for extended periods. Replication offers fault tolerance against a single database server loss and enhances data reliability.

For this application, CSDs offer in-storage processing capability, which allows for horizontal scaling and high availability in a single machine. As depicted in Fig. 5.2, using two CSDs can emulate a data-storage server within another storage server, thereby reducing the number of server deployments from three down to only one. This approach significantly reduces resource utilization for each cluster, while the use of multiple replicas per storage server maximizes redundancy.

5.3.4 Sharding and Replication

Sharding and replication are considered to be the most effective configurations for enhancing performance and ensuring data security. Traditionally, the use of two storage servers per replica set and one or more servers per shard was necessary to facilitate these functions. However, with the introduction of CSDs, this requirement is no longer applicable, and it is now possible to configure



(a) CSD Configuration **(b) Standard Configuration**

Figure 5.2: Data storage servers can be substituted by CSDs in replication mode.

either or both functionality into a single server. By utilizing a single storage server with multiple shards, the data center's physical footprint and overall cost can be reduced, while providing better performance per host and lower replication latency. Fig. 5.3 illustrates a sharding and replication scenario that employs CSDs and has three shards and two replicas for each shard.

5.4 Summary

In this chapter, we discussed several applications that can greatly benefit from distributed, near-data processing on a group of CSDs. A common feature of all these application is that they are more communication-intensive, rather than compute-intensive. This feature aligns well with the CSD concept, as CSDs have a great advantage on the I/O communication side, while lacking powerful processing capabilities.

Given NLP inferencing as an example, many NLP applications require huge I/O transfers along with a small computation to generate output. Our solution is to run the NLP inferencing on CSD with higher I/O speed and lower processing power, rather than transferring large tables of data to

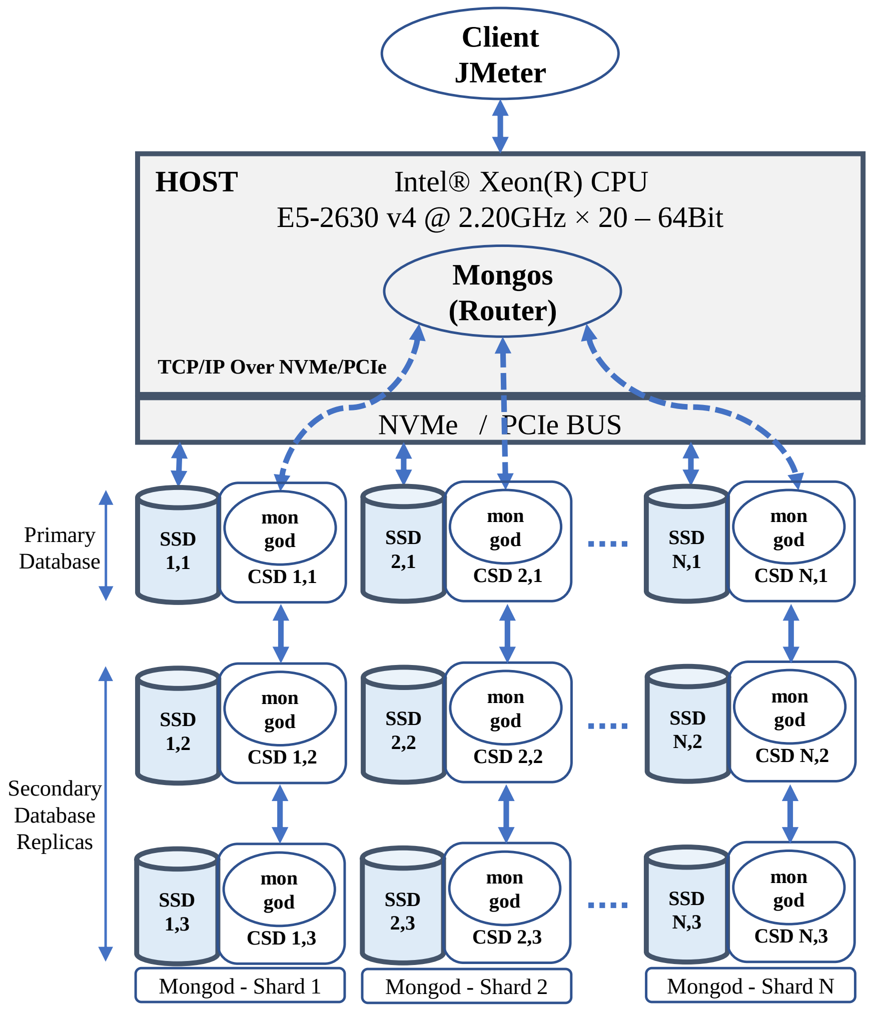


Figure 5.3: Sharding and replication scenario of three shards and two replicas for each shard based on CSDs.

through a (relatively) slow transmission channel and to a powerful processor on the host, where the high processing power might not be useful.

Another example of applications that can benefit from CSD design is the database management applications such as MongoDB. CSD enables running database management agents inside the storage drive, enabling each CSD to respond to the queries on their own and directly provide the data to the requesting party. In the next chapter, we will see these applications in action and examine how efficient CSDs can be in an orchestrated environment.

Chapter 6

Experimental Results

In this chapter, we review our hardware setup and how we prepared our experiments to benchmark both the CSDs and our developed applications. We start by briefly explaining the server units we used for different tests, along with their hardware and software specs. Then, we presentf the experiment design and the measured results and investigate the effect of CSD on the overall performance of the system.

6.1 System Setup

To evaluate our CSD designs, we choose three data servers that support all three form factors. All servers run the Ubuntu 18.04.6 operating system.

6.1.1 U.2 Form Factor

For the U.2 form factor, we use an AIC 2U-FB201-LX server with an Intel Xeon Silver 4108 CPU with 32 GB of DRAM and 24 Newport CSDs, each with 32 TB of flash memory, making a 2U-class

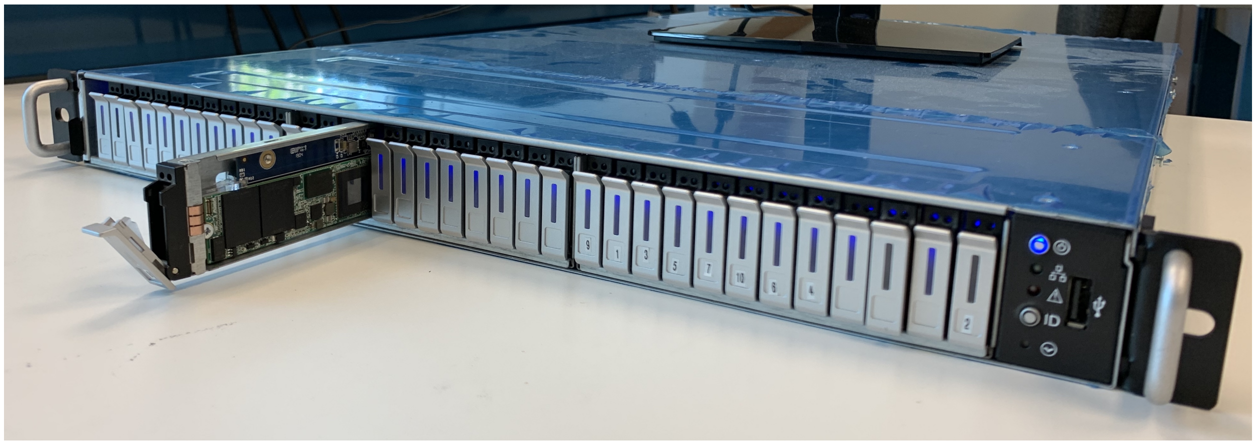


Figure 6.1: A data storage server with 36 Laguna CSD in M.2 form factor.

storage server with a total capacity of 768 TB. A second AIC server with an Intel Xeon Gold 6240 CPU and 96 GB of DRAM is used to evaluate the database benchmarks.

6.1.2 M.2 Form Factor

For the M.2 form factor, we choose a FlacheSAN1N36M-UN server equipped with the same Intel Xeon Silver 4108 CPU and 64 GB of DRAM. We put 36 Laguna CSDs, each with 8 TB capacity on the server to make a 1U class server with 288 TB storage capacity, as shown in Fig. 6.1.

6.1.3 E1.S Form Factor

For the E1.S form factor, we use an AIC FB128-LX equipped with the same Intel Xeon Silver 4108 CPU running at 2.1 GHz and 64 GB of DRAM. This server can support up to 36 E1.S drives in the front bay, 12 TB each, for a total capacity of 432 TB on a 1U class server. Table 6.1 shows the overall spec of the system setup for running the tests.

To measure the power and energy consumption for each test, we use an HPM-100A power meter that sits between the power plug and the server and measures the power consumption of the entire

Table 6.1: Specifications of the servers used for the experiments.

Brand	Model	Storage bay	CPU	DRAM	Dimensions	Max storage capacity
AIC	2U-FB201-LX	24 × U.2	Intel Xeon Silver 4108	32 GB	41.3"×23.4"×12.5"	24×32 TB = 768 TB
AIC	2U-FB201-LX	24 × U.2	Intel Xeon Gold 6240	96 GB	41.3"×23.4"×12.5"	24×32 TB = 768 TB
AIC	FB128-LX	36 × E1.S	Intel Xeon Silver 4108	64 GB	31.5"×17.2"×1.7"	36×12 TB = 432 TB
EchoStreams	FlacheSAN1N36M-UN	36 × M.2	Intel Xeon Silver 4108	64 GB	27.5"×19"×1.75"	36×8 TB = 288 TB

system, including the power of the host processor unit, the storage systems, and peripherals such as the cooling system. Since there is no comparable storage drive on the market with similar capacity and form factor, in all tests except one, we choose the baseline test system to be the server with the same CSD drives but with the ISP engines disabled, acting solely as a storage drive.

6.2 Distributed Training

In this section, we conduct a comprehensive evaluation of both our hardware infrastructure and software framework employed in the distributed training paradigm. We will focus on benchmarking the performance enhancements derived from the deployment of CSDs, examining their impact on the speed and energy consumption aspects of the system.

6.2.1 Stannis

To evaluate Stannis, we use a dataset of 72,000 images as public data and 12000 images as private data distributed over 24 Newport CSDs. We choose MobileNetV2 with 3.47 million parameters and 56 million multiply-and-accumulate (MAC) operations as our main neural network. To compare the speedup on different neural networks, we run the same test for NASNet, InceptionV3, and SqueezeNet. The only concern in choosing a network and the batch size is the available DRAM on the systems. A large batch size on big networks can saturate the DRAM and thus stall the entire training process. The 6 GB DRAM on Newport has proved sufficient for most of the test cases. The solution to DRAM saturation is to choose a smaller batch size. Since the processing speed

Table 6.2: Parameter tuning from algorithm 1

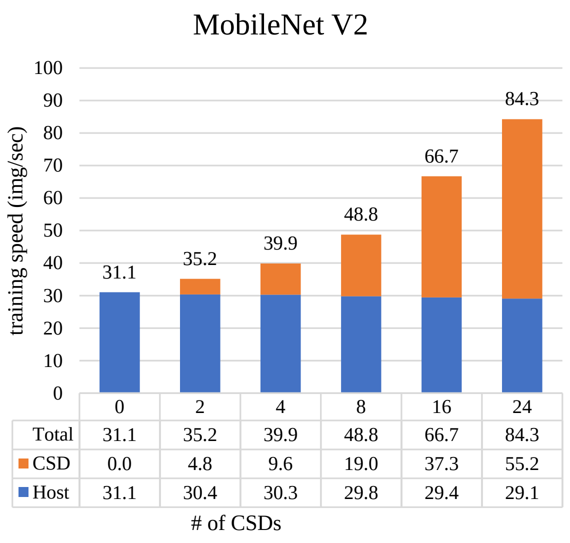
Network	**Param**	**Flop**	**MAC**	**batch size Host / CSD**	**speed (img/sec) Host / CSD**
MobilenetV2	3.47M	7.16M	56M	315 / 25	31.05 / 3.08
NASNet	5.3M	10.74M	564M	325 / 15	47.31 / 2.80
InceptionV3	23.83M	47.82M	5.72G	370 / 16	30.80 / 1.85
squeezenet	1.25M	2.46M	861M	850 / 50	219.0 / 16.3

converges beyond a certain batch size, this reduction in batch size would not affect the processing speed. For instance, the speed for MobileNetV2 on Newport is about 3 images per second for all batch sizes greater than 16. This happens as a result of the full utilization of the processing engine when the task becomes computation-intensive rather than communication-intensive.

Stannis first determines the optimal batch size for the host and the CSDs by running the tuning algorithm for each network. The tuning results are presented in Table 6.2. After tuning, it runs the main training session on different numbers of CSDs.

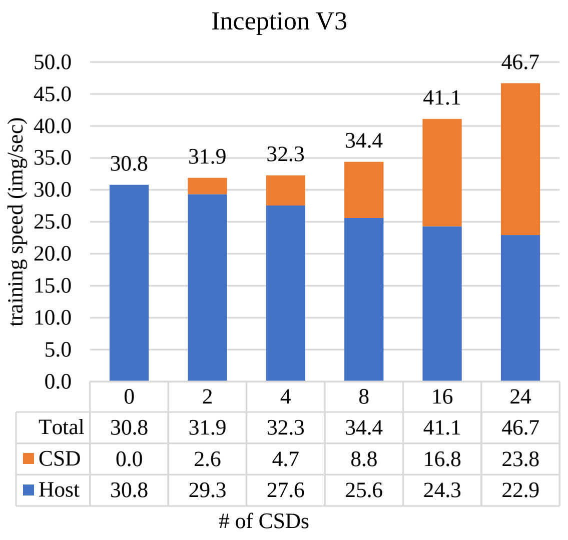
Fig. 6.2 shows the processing speed for different numbers of CSDs for each network training session. There is a slowdown in all processing nodes in distributed training mode, which is due to partial stalls when nodes are synchronizing the parameters. The slowdown pace fades out, and the individual node's performance converges to a certain speed after the number of nodes grows beyond 5-6 devices. The relative speedup for different networks is also shown in Fig. 6.3. The measurements show that smaller networks get better speedup than the larger ones do, because the more parameters to update, the more time it takes to synchronize the nodes. Another important parameter is the number of MACs. As Fig. 6.3 shows, SqueezeNet (2.46M Flops) gets less speedup compared to MobileNetV2 (7.16M Flops), because it has 15× more MACs.

It is difficult to compare the power consumption of our server and a similar setup, but with a non-CSD drive, as there is no equivalent product with the same storage capacity in the market. The closest product to Newport that we could find was the 11-TB Micron MTFDHAL11TATCW-1AR1ZAB SSD. We evaluate the power consumption of an AIC server with 24 11-TB Micron SSDs



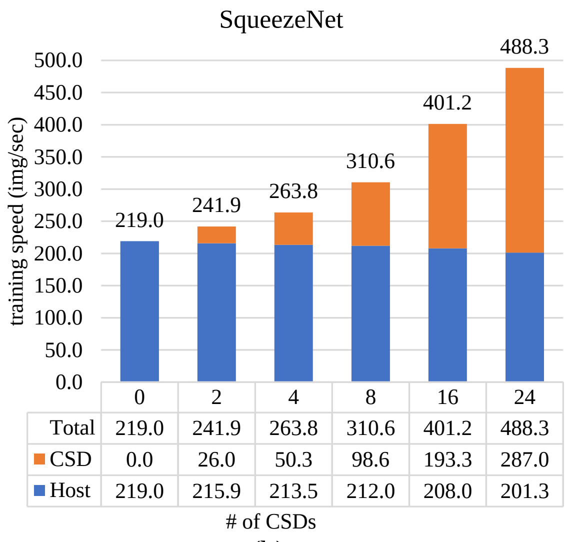
# of CSDs	0	2	4	8	16	24
Total	31.1	35.2	39.9	48.8	66.7	84.3
🟧 CSD	0.0	4.8	9.6	19.0	37.3	55.2
🟦 Host	31.1	30.4	30.3	29.8	29.4	29.1

(a)



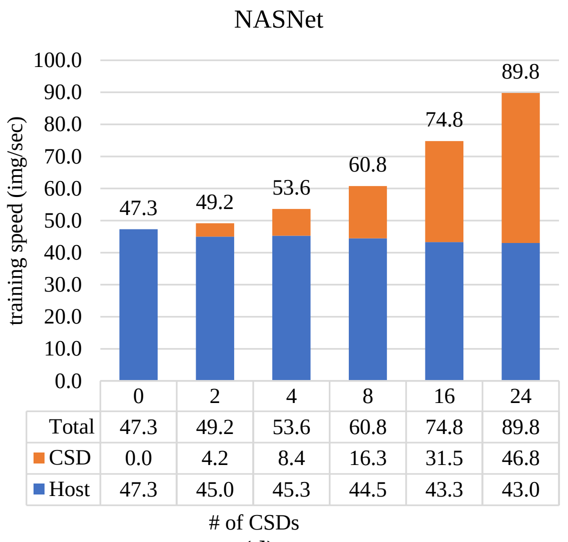
# of CSDs	0	2	4	8	16	24
Total	30.8	31.9	32.3	34.4	41.1	46.7
🟧 CSD	0.0	2.6	4.7	8.8	16.8	23.8
🟦 Host	30.8	29.3	27.6	25.6	24.3	22.9

(b)



# of CSDs	0	2	4	8	16	24
Total	219.0	241.9	263.8	310.6	401.2	488.3
🟧 CSD	0.0	26.0	50.3	98.6	193.3	287.0
🟦 Host	219.0	215.9	213.5	212.0	208.0	201.3

(b)



# of CSDs	0	2	4	8	16	24
Total	47.3	49.2	53.6	60.8	74.8	89.8
🟧 CSD	0.0	4.2	8.4	16.3	31.5	46.8
🟦 Host	47.3	45.0	45.3	44.5	43.3	43.0

(d)

Figure 6.2: Experimental results for distributed training for different NNs.

Table 6.3: Energy consumption of distributed training using Stannis

Number of CSDs	**0**	**4**	**8**	**16**	**24**
Energy per image (J)	13.10	8.30	6.84	5.05	4.02
Energy saving (%)	0%	37%	48%	62%	69%
FLOPS per watt	5.87M	7.05M	8.18M	10.37M	12.26M

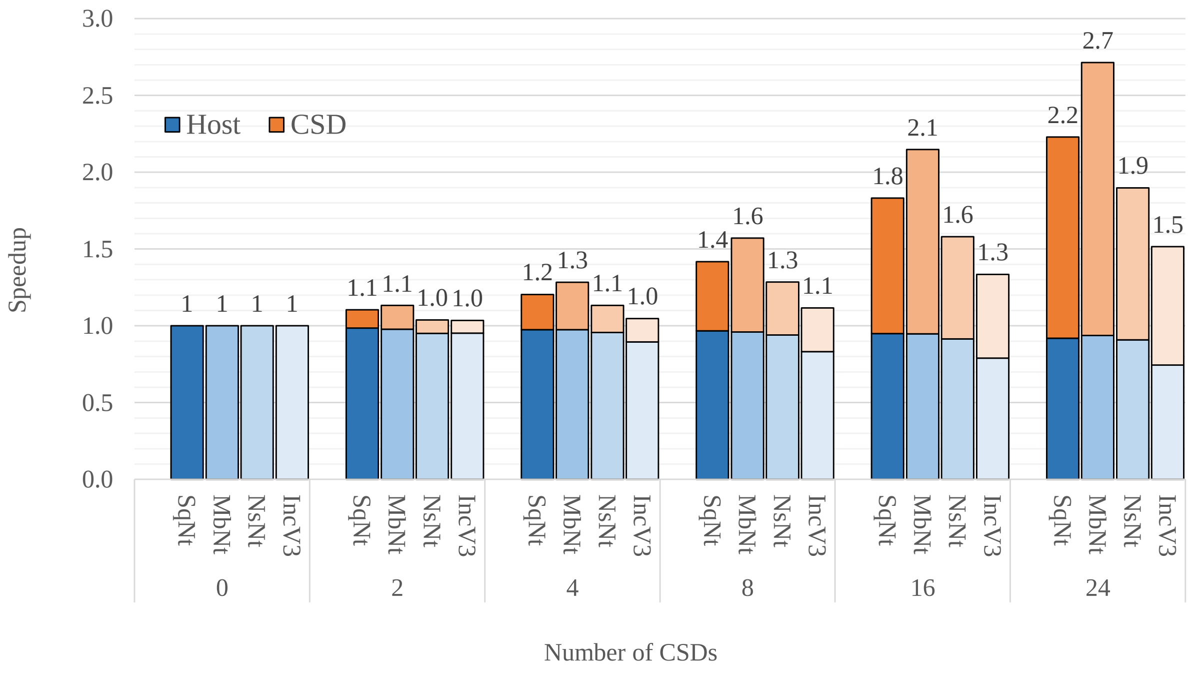


Figure 6.3: Normalized results of distributed training for different NNs.

against the same system with 24 32-TB Newport CSDs. Table 6.3 shows the energy per processed image for MobileNetV2. Measurements show up to 69% saving in energy per processed image and 2× Flops per watt with 24 Newport CSDs compared to a system with no CSD. We skip the results for other networks since the measurements are almost identical.

6.2.2 HyperTune

To evaluate the HyperTune algorithm, we run a training session on three similar nodes (AIC 2U-FB201-LX servers) in terms of processing performance, as similar processing performance highlights the significance of HyperTune. We choose to run the tests on the MobileNetV2 neural network, with 300,000 images as the input to the system. We simulate external workloads using the Gzip compression application, which enables the desired number of cores on the processors to be occupied. Without loss of generality, we also generate interrupts to one node at a time to facilitate the experimental procedure: since the master node collects the interrupt reports from all nodes, it can easily detect if the slowdown is due to the local workload or an external node.

Similar to the previous experiment, Stannis starts by finding the best batch size for each node, which is 180 for all three nodes, since they all use the same Intel Xeon processor. Then, in two separate steps on separate nodes, we set Gzip to occupy 4 and 6 cores out of 8 cores and monitor the speed change. We define the performance as the average number of processed images per second for the training. As Fig. 6.4 shows, the overall processing speed of the three nodes during the test is 93.4 images/sec in normal operation mode. As the workload increases, the speed drops to 75.6 and 53.3 images/sec for 50% and 75% external workloads, respectively. These speeds remain nearly constant as long as the workload sustains thereafter. In contrast, if HyperTune detects interrupts by a new workload and determines a non-transient decline in the speed, it recomputes the new batch size for the busy node and updates the hyperparameters, which are 140 and 100 for 4-core and 6-core workloads, respectively. By changing the batch size, the other two nodes regain the initial processing speed, which shows that the algorithm works effectively in determining the new batch size. The processing speed declines respectively to 83.7 and 85.8 images/sec for 75% and 50% external workloads. That is, HyperTune is 14% and 57% faster than the baselines at no cost of power or accuracy and with negligible computation overhead.

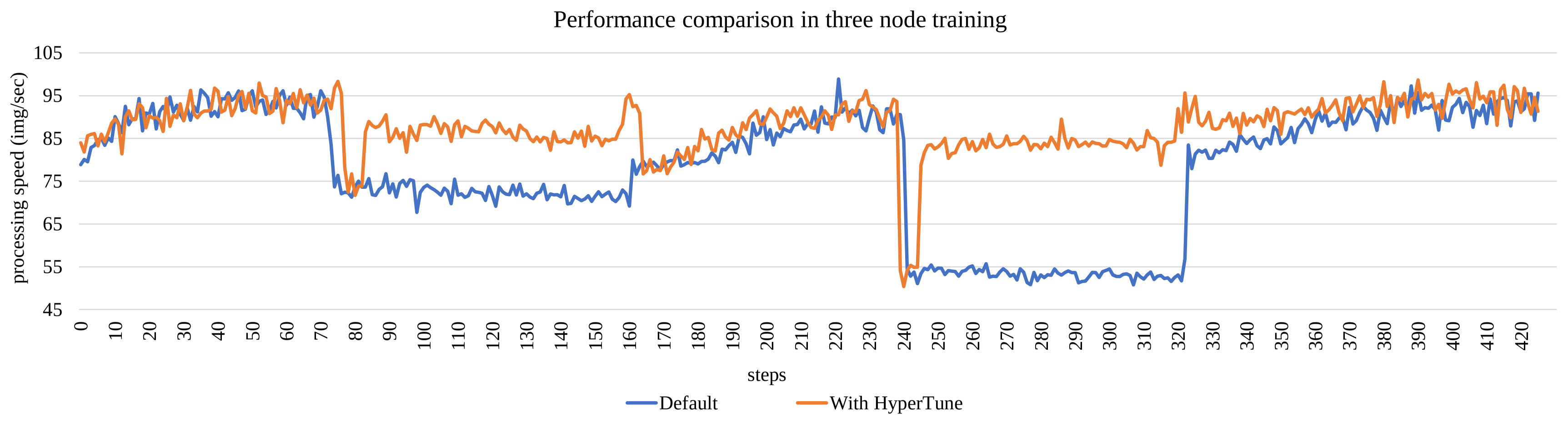


Figure 6.4: Experimental results for Hypertune.

Table 6.4: Summary of the NLP applications characteristics.

application	input type	input size	model size	processing time (base 100 us)
sentiment analysis	text	small	1.5 MB	1×
movie recommender	text	very small	3.4 GB	20×
speech to text converter	audio	large	50 MB	1000×

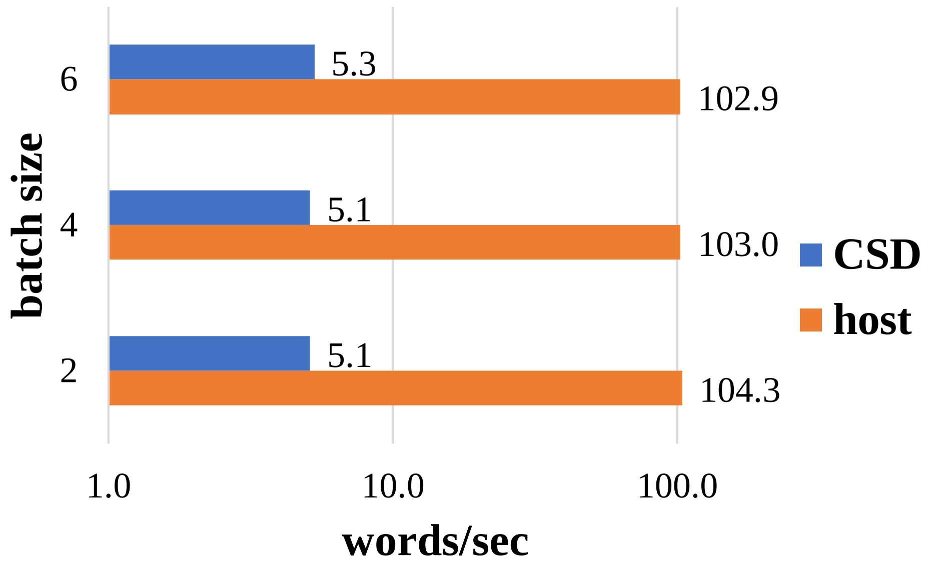
6.3 Distributed Inferencing - NPL

We have chosen three common NLP applications, namely sentiment analysis, movie recommender, and speech-to-text converter to run on our server. Although the models are intentionally chosen in a way that fits in the CSD's internal memory, they cover a wide range of characteristics in terms of model size, input size, and input type. A summary of all three applications are presented in Table 6.4.

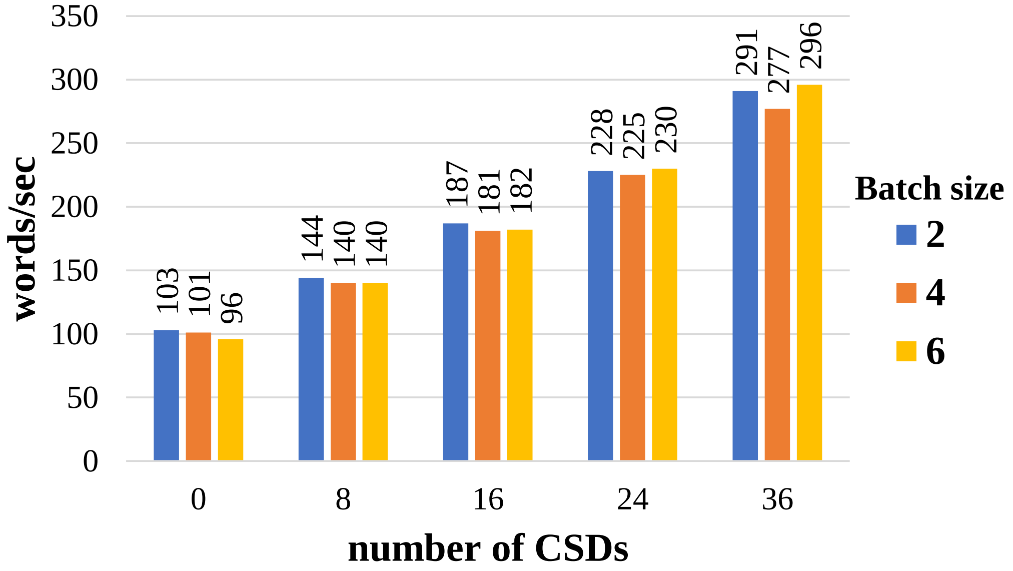
6.3.1 Speech-to-text Benchmark

This benchmark is developed based on Vosk, an offline speech-recognition toolkit. Vosk can support 17 different languages and has multiple models, as small as 50 MB, that can be deployed on ARM-based and other lightweight devices [86]. To test our speech-to-text benchmark, we use a public-domain speech dataset named LJ [87], which consists of 13,100 short audio clips of a single speaker reading passages from seven non-fiction books. The dataset is about 24 hours long in 225,715 words. When running a small benchmark, the host can process 102 words/sec, whereas a single-node CSD can do 5.3 words/sec. This batch size ratio of around 20 depends on the nodes' processing performance and is almost the same for all three applications, as their common bottleneck is the computation. We use this ratio in the scheduler.

Fig. 6.5b shows the overall output of this benchmark in terms of the number of transcribed words per second, based on the batch size and the number of engaged CSDs. It shows that by augmenting



(a)



(b)

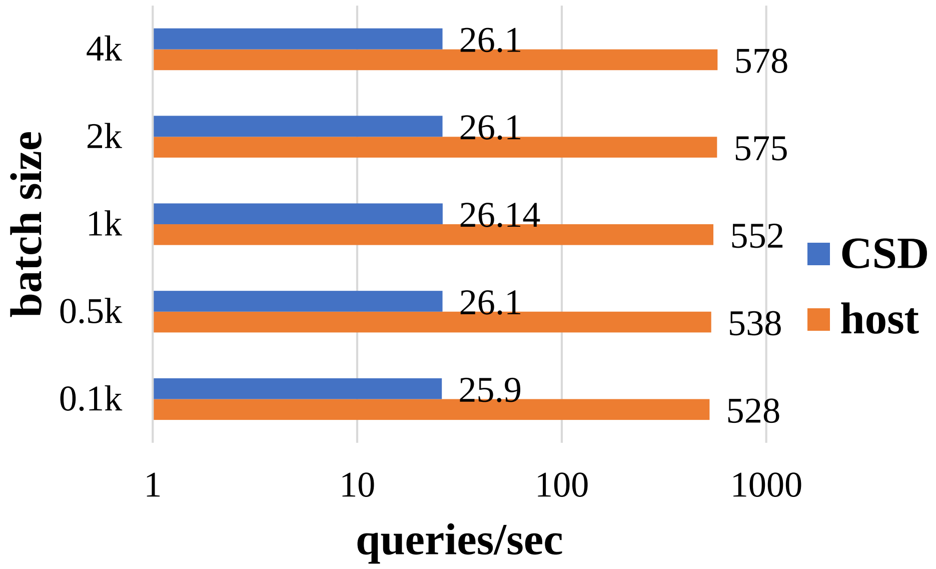
Figure 6.5: (a) Individual node's benchmarks and (b) performance results for the speech-to-text application.

the host with the processing power of all 36 CSDs, the output rate increases from 96 words/sec to 296 words/sec for a batch size of 6. That is 3.1× better performance compared to the host alone. Also, the single node performance for different batch sizes shown in Fig. 6.5a means that the processing speed does not change much when varying the batch size. In terms of I/O transfers, CSD engines process 68% of the input data ($(296 - 96) \div 296 \simeq 0.68$); in other words, 2.58 GB out of the 3.8 GB of the dataset never leave the storage units, and the only output of about 1.2 MB transferred to the host is the output text. This reduction in the number of I/O's and in data transfer size can drastically reduce the power consumption, network congestion, and usage of the host's CPU memory bandwidth. It also enhances the security and privacy aspect of the data, as it never leaves the storage device.

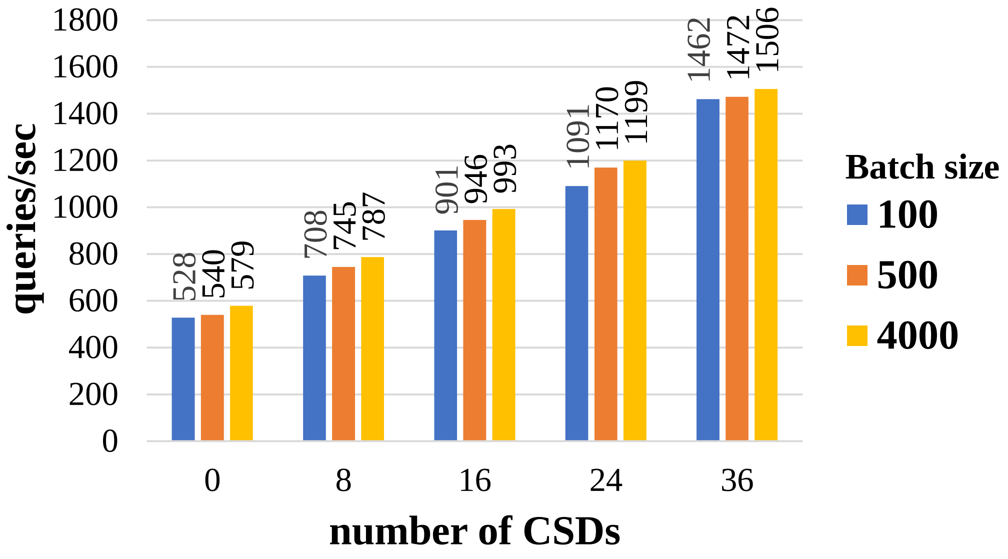
6.3.2 Movie Recommender Benchmark

Our second benchmark is a movie recommendation system based on [88]. The recommender creates a new metadata entry for each movie, including the title, genre, director, main actors, and storyline keywords. It then uses the cosine-similarity function to generate a similarity matrix to be used later for content suggestions. An extra step uses the ratings and the popularity indexes to filter the top results. For each query, the target movie title is sent to the recommender function, which returns the top 10 similar movies. To train and test the application, we use the MovieLens Dataset [89], which consists of 27 million ratings and 1.1 million tag applications applied to 58,000 movie titles by 280,000 users. We run the training process once and store the matrix on Flash for further use. To simulate the queries, we make a list of all movie titles and randomly shuffle them into a new list. The results from Fig. 6.6a show that, just like the speech-to-text application, the processing speed does not change much (about 3%) over different batch sizes. Therefore, smaller batch sizes are preferable due to the shorter worst-case latency.

Fig. 6.6b shows the result of movie-recommendation queries for different configurations. With 36



(a)



(b)

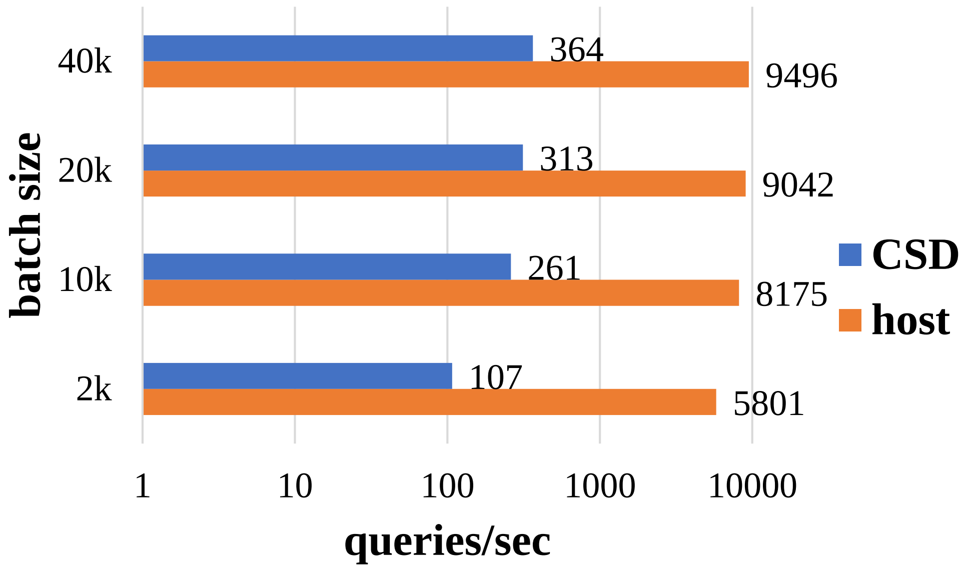
Figure 6.6: (a) Individual node's benchmarks and (b) performance results for the movie recommendation application.

drives, the system can address 1506 queries/sec compared to the 579 queries/sec of the standalone host, which is 2.6× improvement. The measurements show that the processing speed scales linearly with the number of CSDs, and the overhead of distributing the task is almost zero. In fact, in some cases, the distributed result scales superlinearly due to the faster data access speed of the ISP engine than on the host. In the host-only configuration, the host's CPU never reaches 100% utilization due to other I/O bottlenecks. By increasing the number of CSD drives and further distribution of input data, the I/O bandwidth increases, shifting the bottleneck on the host to the computation, which in turn pushes the CPU to higher utilization levels and enhances the host output. This is how the added processing capability from the new CSDs results in a benevolent cycle.

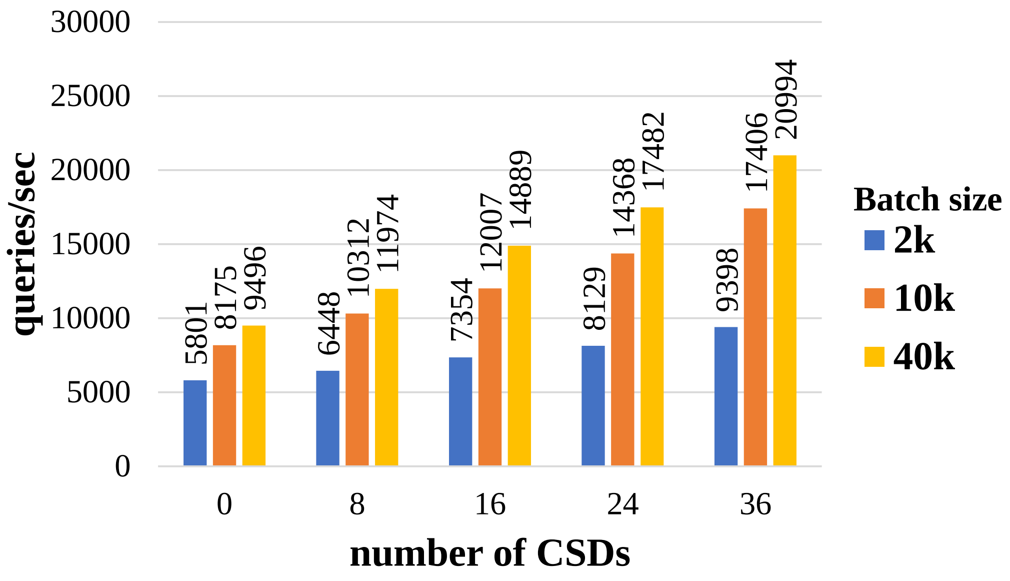
6.3.3 Sentiment Analysis Benchmark

The last benchmark is a tweet sentiment-analysis app based on Python's Natural Language Toolkit (NLTK). We modify this app from [90] to fit to our parallelization goals. It uses labeled data to train a model to detect the positivity or negativity of the tweet's content and then uses the model to predict the sentiment of the incoming tweets. The input data undergoes a series of preprocessing steps where the words are tokenized, lemmatized, stripped of noises and meaningless words, and eventually converted to a dictionary of words ready to be fed to the model. Our test dataset consists of 1.6 million tweets [91] and is duplicated in cases where we need a larger number of queries. We run a single-node test to evaluate the host and the CSD's performance for different batch sizes. Unlike the other two benchmarks, performance changes considerably based on batch size.

Fig. 6.7a plots the performance in terms of queries per second for different batch sizes on a logarithmic scale. It shows that both CSD and the host get better performance with larger batch sizes. For instance, CSD's performance increases from 107 queries/sec for the batch size of 2k to 364 queries/sec for the batch size of 40k, which is a 3.4× increase. However, the trade-off for large batch sizes is the increased latency in processing some of the queries. Because the input is



(a)



(b)

Figure 6.7: (a) Individual node's benchmarks and (b) performance results for the sentiment analysis application.

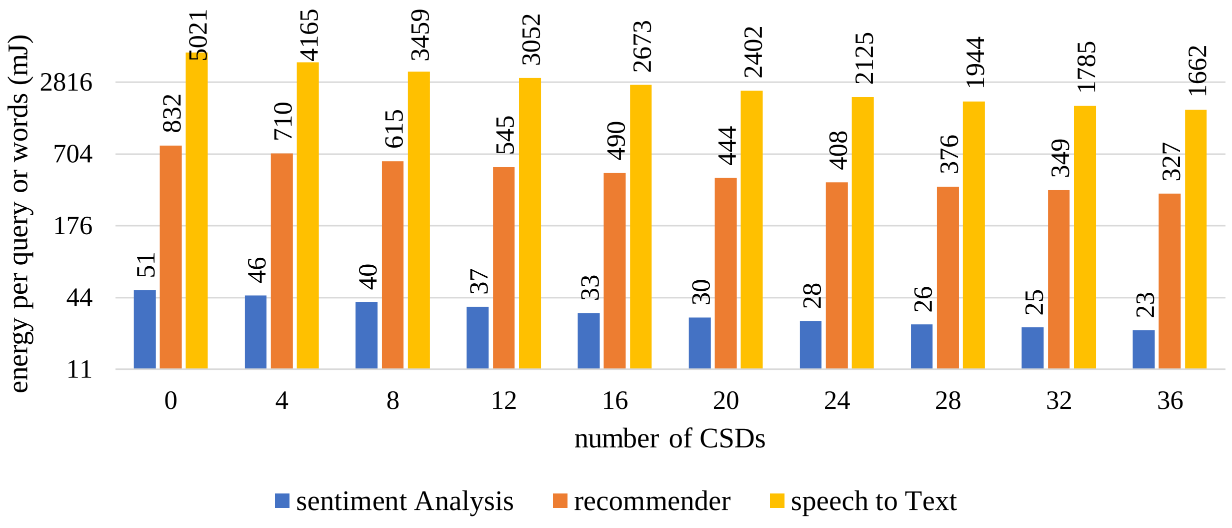


Figure 6.8: Energy consumption per query for different configurations of the three applications.

Table 6.5: Summary of the experimental results.

test case	speech to text	recom-mender	sentiment analysis
accuracy	same	same	same
max speedup	3.1×	2.8×	2.2×
energy per query (host) (mJ)	5021	832	51
energy per query (w/CSD) (mJ)	1662	327	23
energy saving per query (%)	67%	61%	54%
data processed on host (%)	32%	36%	44%
data processed in CSDs (%)	68%	64%	56%

sequentially fed to the nodes, once data is assigned to one processing node, it has to wait for prior queries on the same node to finish but cannot be migrated to run on some other idle nodes. For example, if the first N queries are assigned to node 1 and the next M queries $(N+1 \ldots N+M)$ are assigned to node 2, then the N^{th} query has to wait for all queries from 1 to $N-1$ to finish before it can be processed on node 1; whereas in smaller batch sizes, those queries could be assigned to other available nodes. Based on these numbers, we set the batch size ratio to $9496 \div 364 \simeq 26$ and run a benchmark with 8 million tweets. Fig. 6.7b shows the performance for different batch sizes and on different numbers of CSDs. The best result is for the batch size of 40k, where the number of queries per second increases from 9496 to 20994, or 2.2× the performance.

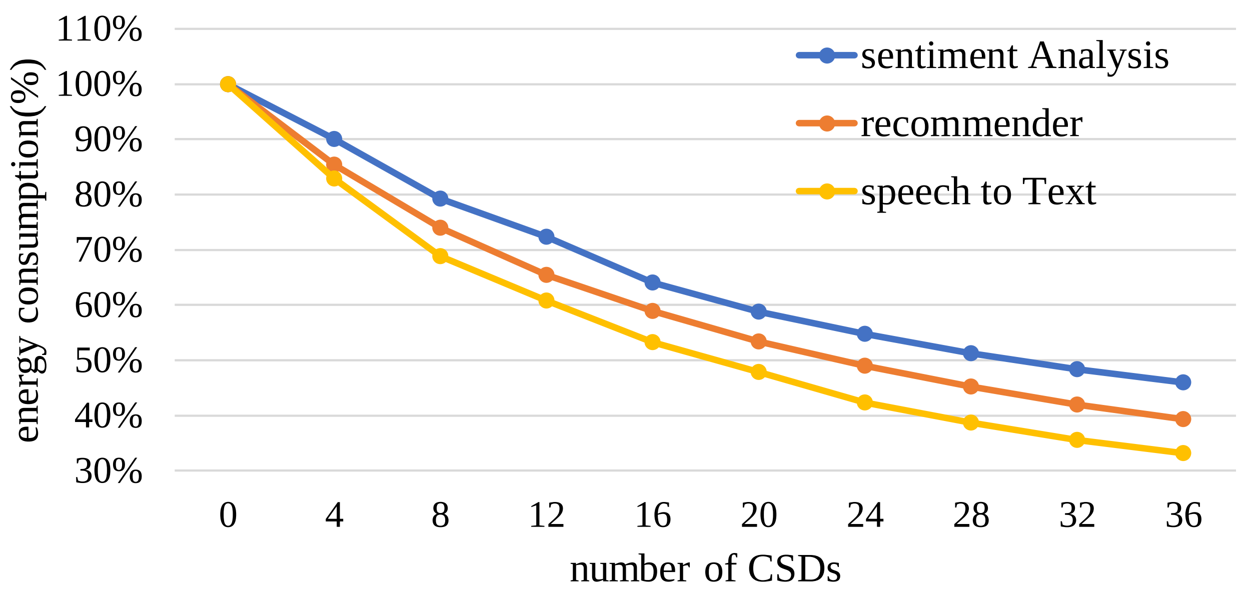


Figure 6.9: Energy per query, normalized to host-only setup.

6.3.4 Power and Energy Analysis

For all NLP test cases and in idle mode, the server consumes 167 W with no storage drives and 405 W with 36 CSDs. Thus, each CSD consumes an average of 6.6 W, compared to 10-15 W for commercial E1.S units with lower storage capacity in normal operation mode. When we run the benchmarks, the power consumption of the entire system goes up to 482 W without enabling ISP (i.e., CSD acting as storage only), compared to 492 W with all 36 ISP engines running. That is, each ISP engine consumes 0.28 W on top of the storage-only feature. Note that our datasets are small in size and all three benchmarks are compute-bound rather than I/O-bound, which means most of the power is drawn for CPU activity. As a result, all three benchmarks yield the same power measurements, and only the amount of energy per query varies by application. Fig. 6.8 shows the energy consumption per query, or per word in the case of the text-to-speech benchmark. Fig. 6.9 shows the normalized energy consumption to better demonstrate the energy saving trend. Note that these measurements are averaged over the entire test session. The power consumption enhancement can be attributed to a) the low-power processors of the ISP, and b) the reduction in data transfer size, as minimal raw data needs to be moved out of the storage drives. For future work, we plan to measure the energy consumption in each step to better demonstrate the importance of data transfer reduction. Finally, Table 6.5 summarizes the experimental data for all three applications.

6.4 Database - MongoDB

Database benchmarks tend to face different bottlenecks and can be tricky. To overcome these bottlenecks over different configurations, we investigate two scenarios for benchmarking: direct configuration and sharding clustering configuration. For database benchmarking and data generation, we used Yahoo! Cloud Serving Benchmark (YCSB) [92] suite, which allows measuring the performance of both NoSQL and SQL database management systems with simple database operations on synthetic data. Since YCSB requires considerable resources and power to generate data requests, we use a separate system (HP ProLiant DL380p Gen8) solely to generate the benchmarks and queries. All systems are connected through a regular 1-GBps router. Note that we use the same parameters for YCSB in all cases (record count = 100k, operation count = 100k, read operation = 0.5, write operation = 0.5).

6.4.1 Direct Configuration

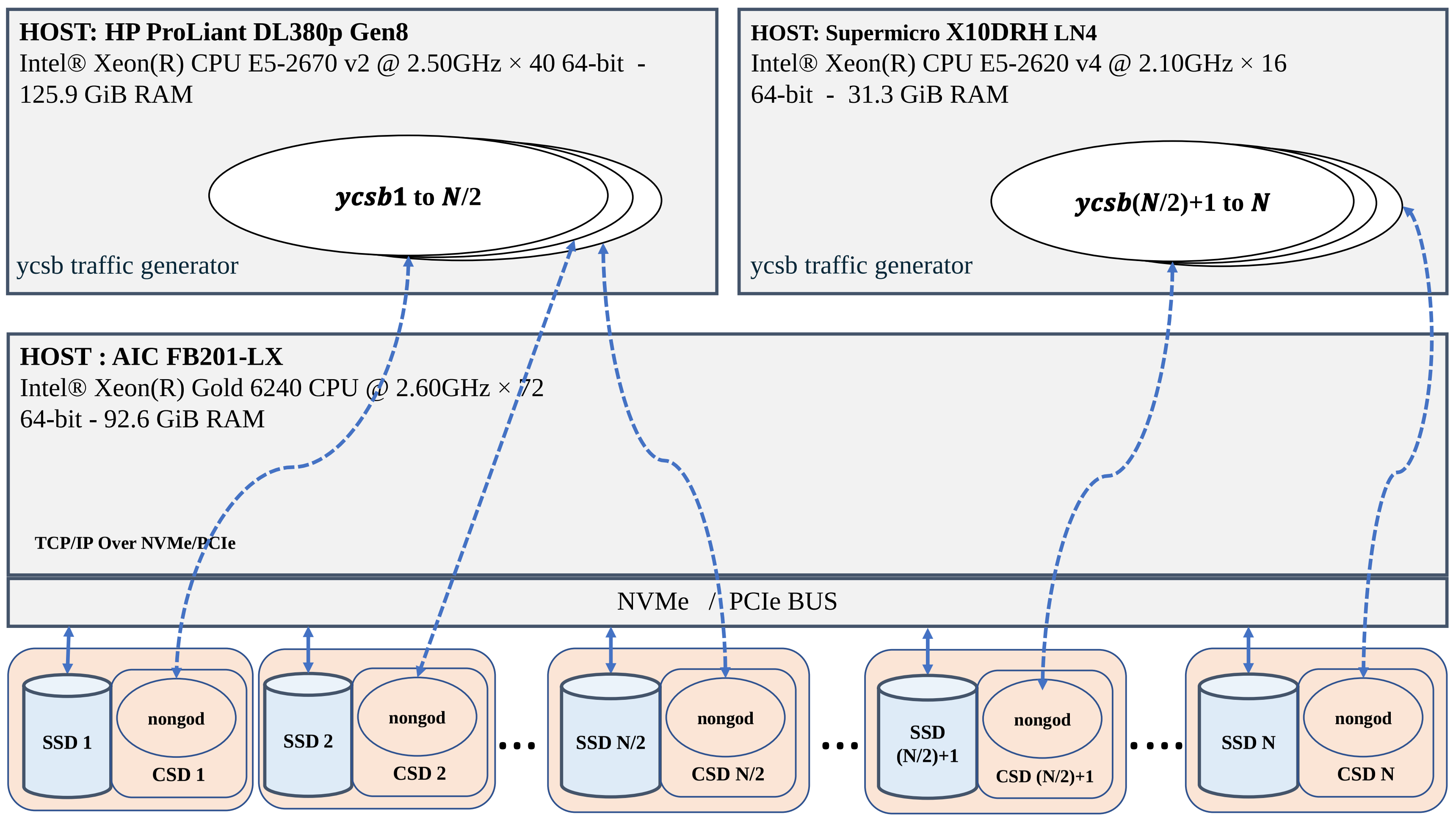

HOST: HP ProLiant DL380p Gen8
Intel® Xeon(R) CPU E5-2670 v2 @ 2.50GHz × 40 64-bit - 125.9 GiB RAM
ycsb1 to N/2
ycsb traffic generator
HOST: Supermicro X10DRH LN4
Intel® Xeon(R) CPU E5-2620 v4 @ 2.10GHz × 16 64-bit - 31.3 GiB RAM
ycsb(N/2)+1 to N
ycsb traffic generator
HOST : AIC FB201-LX
Intel® Xeon(R) Gold 6240 CPU @ 2.60GHz × 72 64-bit - 92.6 GiB RAM
TCP/IP Over NVMe/PCIe
NVMe / PCIe BUS
SSD 1
nongod
CSD 1
SSD 2
nongod
CSD 2
SSD N/2
nongod
CSD N/2
SSD (N/2)+1
nongod
CSD (N/2)+1
SSD N
nongod
CSD N

Figure 6.10: Direct mode configuration in MongoDB.

Fig. 6.10 depicts the direct mode configuration, where each CSD acts as a standalone database and queries are sent directly to each CSD. As seen earlier, ISP provides the opportunity to run multiple databases on a single data server, i.e., each database runs an independent MongoDB agent while the host system runs its own agent, all on the same CSD. Since a single database could not saturate the storage nodes, we put two databases on each CSD, one managed by the ISP engine and the other by the host.

Fig. 6.11 shows the overall throughput in terms of operations per second. As can be seen, the increase in the host performance is sub-linear but monotonic, whereas the utilization of each CSD degrades as the number increases.

Although our experimental server maxed out at 24 CSDs due to the inherent technical limitations of the storage server design, the utilization can go higher by deploying more CSDs. However, based on the measurements, we estimate the scaling to be upper-bounded at around 56 drives. Compared to a data server with the same configurations but with regular SSDs, the overall throughput of the system with 24 CSDs is higher by 2.78×.

To investigate the power and energy consumption, we measure the overall power consumption of the server with 20 CSDs in idle, host-only, CSD-only, and hybrid modes. As Fig. 6.12 shows, when running the application on the host-only setup with 20 CSDs with inactive ISPs, the total power consumption rises from 354 W to 456 W, yielding an average energy consumption of 798 uJ per operation. For the CSD-only case, where the host processor is inactive, the increase in power consumption is only 29 W for the entire server, which yields an average of 416 uJ per operation. When both CSD and host are active, the overall power goes up to 421 W, and the average energy consumption drops to as low as 339 uJ per operation. These results show that deploying CSDs for I/O-intensive applications can greatly reduce energy consumption by up to 57% while increasing performance by up to 3.4×.

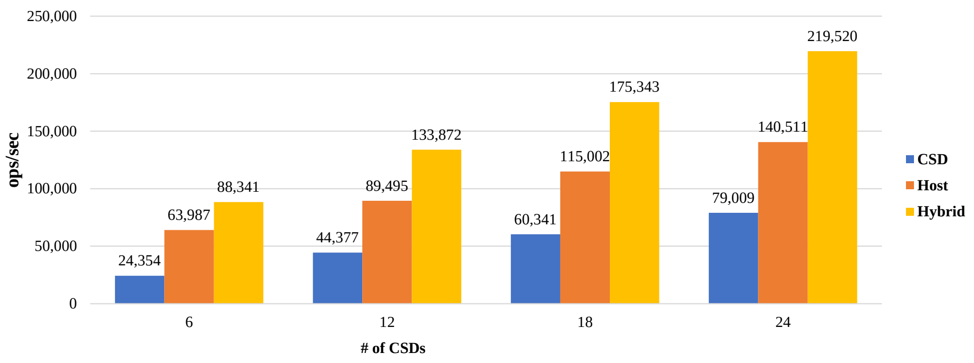


Figure 6.11: Experimental results for MongoBD in direct mode.

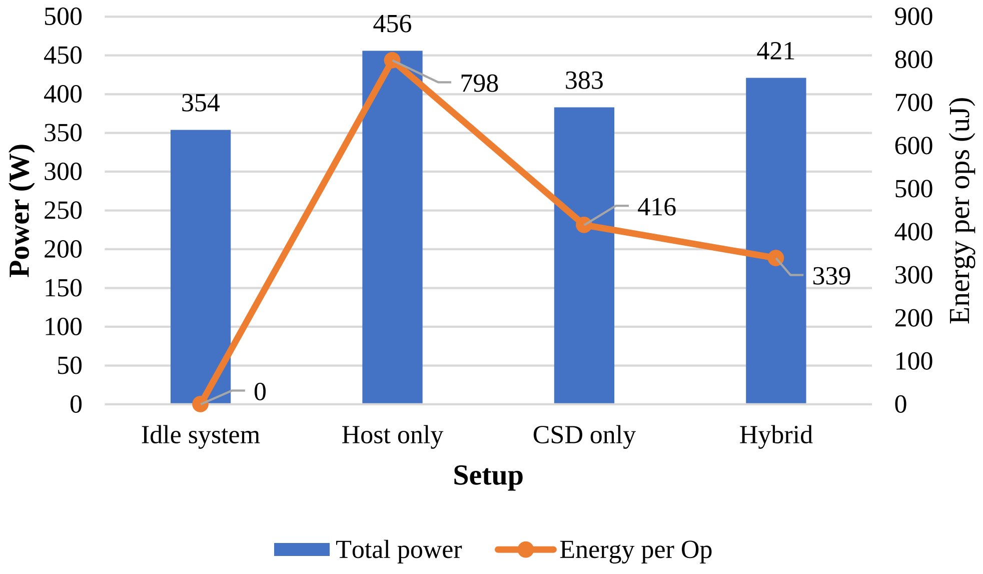


Figure 6.12: Power analysis of MongoDB in direct mode.

6.4.2 Sharded Cluster Configuration

In a sharded cluster, each database is sharded over several storage nodes, or CSDs in our case. To handle data forwarding, a set of MongoDB routers, known as *Mongos*, are needed to run on the host system. These routers redirect queries to the appropriate storage node. Fig. 6.13 shows the system configuration for sharded cluster test. Each router connects to a MongoDB agent that either runs on the host or on one of the CSDs. Note that just like the previous test, two external systems run the YCSB and generate the queries. Fig. 6.14 shows the throughput of the sharded cluster with 24 CSDs for host-only, CSD-only, and hybrid modes. Since the router application is lightweight and the overhead is negligible, the host's performance remains the same after engaging the CSDs. Each CSD handles an average of 765 operations per second. That said, our data storage server with 24 CSDs can increase the performance by up to 1.7 times compared to the same storage server with regular SSDs.

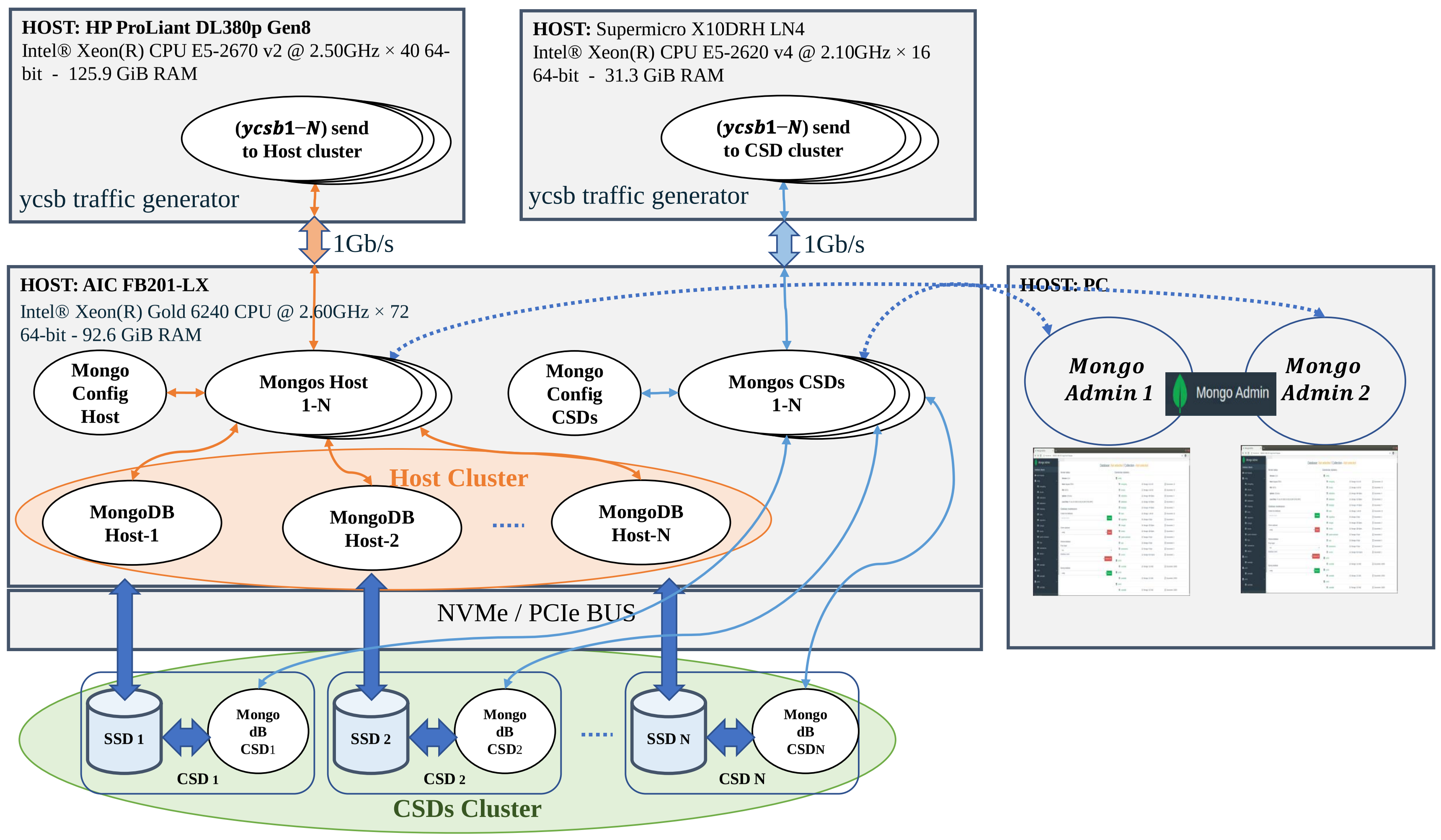


Figure 6.13: Sharded cluster configuration in MongoDB.

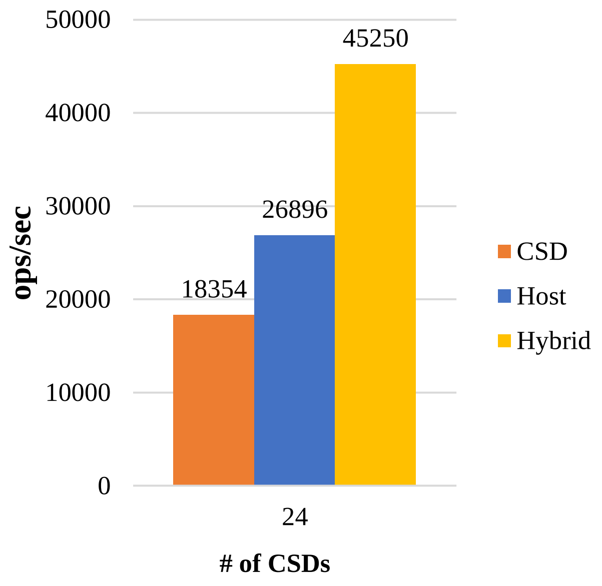


Figure 6.14: Experimental results for MongoBD in sharded cluster mode.

Another observation is the performance drop on each CSD in sharding clustering mode compared to direct mode. While sharding enables handling much larger datasets in a distributed fashion, it can degrade the performance as the routers' operations add latency to the system. We also investigate a sharded cluster with replications, where each data is replicated on multiple CSDs to provide resiliency. In this scenario, the system performance is just like the regular sharded cluster, but with more reliability and less overall capacity, as each data is replicated on several CSDs. We skip the results of sharded and replicated clusters to avoid redundancy with the previous setup.

6.5 Summary

In this chapter, a series of experiments were conducted to assess the efficacy of both our hardware and software solutions. Given that the deployment and utilization of CSD closely resemble that of SSD, requiring no additional setup steps for CSD, our investigation focused on two key performance parameters: processing speed and energy consumption per unit of work. Leveraging the inherent advantage of CSDs, namely their ability to provide rapid and high-speed access to stored data, our benchmarking efforts targeted applications with high communication requirements. The experimental results substantiated that the deployment of CSDs introduces minimal overhead, while significantly enhancing overall system performance. Furthermore, these experiments demonstrated the feasibility of executing these tasks on multiple nodes, with a maximum of 37 nodes observed in our study, and the potential for further expansion. The primary limiting factor in our specific scenario was the maximum number of available slots for inserting CSDs on an individual server. Depending on the application and the number of CSDs deployed, noteworthy improvements of up to 3.1× in processing speed and 67% in energy consumption were observed.

Chapter 7

Conclusions

Computational storage drives (CSD) are a promising building block for servers that run next-generation data-dominated computing applications. They remove the bottleneck of moving huge amounts of data, resulting in significant gains in processing speed and reduction in power. However, hardware and software must work together to achieve the potential benefits.

Our hardware contribution is a new integrated controller-processor ASIC that enables the SSD to run code in-storage with minimal overhead. It runs conventional Linux executables without modification and is more practical than FPGA solutions that require RTL design and reconfiguration, which are difficult to integrate into an OS-based, multitasking server environment. Our ASIC also works well in the storage system environment, where GPU solutions would not work due to the high power demand.

To realize its full potential, the software must identify and overcome synchronization and other limits on parallelization. We proposed Stannis to minimize the synchronization stalling by equalizing the workload distribution for neural-network training applications based on pre-run-time test runs, and our HyperTune mechanism adapts the hyperparameters to keep the workload balanced at runtime. We also developed an online scheduler for distributing inferencing workloads, as represented by several public NLP applications. In the case of databases, the existing sharding structure already implemented in MongoDB can readily take advantage of the CSD organization and benefit from both the power and performance advantages without modification, as we have demonstrated in our experiments.

Directions for future work include methods to handle data-aware and hardware-aware workload distribution as well as their evaluation. First, data-aware distribution will enable better utilization of CSDs by exploiting not only temporal locality but also spatial locality of data, by classifying data requests and queries into categorical groups and redirecting them to associated nodes. Another direction will be to explore hosts with different performance characteristics, including the use of FPGAs and GPUs, in achieving the most streamlined configuration in conjunction with our CSD setups. Finally, a more in-depth comparison with other CSD implementations can better demonstrate the merits of each design.

Bibliography

[1] Jordi TORRES.AI. Scalable deep learning on parallel and distributed infrastructures, Jun 2021. URL `https://towardsdatascience.com/scalable-deep-learning-on-parallel-and-distributed-infrastructures-e5fb4a956bef`.

[2] How data is stored and what we do with it, Nov 2017. URL `https://insidebigdata.com/2017/11/12/how-data-is-stored-and-what-we-do-with-it/`.

[3] Thomas Alsop. Global enterprise byte shipment share: HDD and SSD 2010-2025, Mar 2020. URL `http://www.statista.com/statistics/815308/worldwide-enterprise-byte-shipment-share-hdd-ssd/`.

[4] URL `https://www.trendforce.com/news/2024/04/09/news-ssd-prices-keep-rising/`.

[5] H. Bahn and K. Cho. Implications of NVM based storage on memory subsystem management. *Applied Sciences*, 10:999, 2020.

[6] 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.

[7] Benefits of using NVMe over PCI Express fabrics. URL `https://www.dolphinics.com/solutions/nvme_over_pcie_fabrics.html`.

[8] The Open Compute Project (OCP) web page, . URL `https://www.opencompute.org/`.

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

[10] Seagate Kinetic HDD produce web page. URL `https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/`.

[11] 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.

[12] Advancing storage and information technology. URL `https://www.snia.org/`.

[13] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage processing of database scans and joins. *Inf. Sci.*, 327(C):183–200, January 2016. ISSN 0020-0255. doi: 10.1016/j.ins.2015.07.056. URL https://doi.org/10.1016/j.ins.2015.07.056.

[14] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating external sorting via on-the-fly data merge in active SSDs. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[15] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, 2017. doi: 10.1109/HPCA.2017.15.

[16] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active Flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, San Jose, CA, February 2013. USENIX Association. ISBN 978-1-931971-99-7. URL https://www.usenix.org/conference/fast13/technical-sessions/presentation/tiwari.

[17] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. *RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference*, page 717–729. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383172. URL https://doi.org/10.1145/3445814.3446763.

[18] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016. doi: 10.1109/ISCA.2016.23.

[19] Jasmine OpenSSD platform. URL http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform.

[20] Cosmos OpenSSD platform. URL http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Platform.

[21] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: An appliance for big data analytics. *SIGARCH Comput. Archit. News*, 43(3S):1–13, June 2015. ISSN 0163-5964. doi: 10.1145/2872887.2750412. URL https://doi.org/10.1145/2872887.2750412.

[22] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. SmartSSD: FPGA accelerated near-storage data analytics on SSD. *IEEE Computer Architecture Letters*, 19(2):110–113, 2020. doi: 10.1109/LCA.2020.3009347.

[23] Keith Chapman, Mehdi Nik, Behnam Robatmili, Shahrzad Mirkhani, and Maysam Lavasani. Computational storage for big data analytics. In *Proceedings of 10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19)*, 2019.

[24] 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.

[25] Siavash Rezaei. *Field Programmable Gate Array (FPGA) Accelerator Sharing*. PhD thesis, UC Irvine, 2020.

[26] 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.

[27] Misha Sadeghi, Seyyed Ahmad Razavi, and Morteza Saheb Zamani. Reducing reconfiguration time in FPGAs. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 1844–1848. IEEE, 2019.

[28] Siavash Rezaei, Eli Bozorgzadeh, and Kanghee Kim. Ultrashare: FPGA-based dynamic accelerator sharing and allocation. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5. IEEE, 2019.

[29] Seyyed Ahmad Razavi and Morteza Saheb Zamani. Improving bitstream compression by modifying FPGA architecture. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 167–170, 2013.

[30] Smartssd, May 2021. URL `https://samsungsemiconductor-us.com/smartssd/`.

[31] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. NASCENT: Near-storage acceleration of database sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 262–272, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382182. doi: 10.1145/3431920.3439298. URL `https://doi.org/10.1145/3431920.3439298`.

[32] 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, 2013. doi: 10.1109/MSST.2013.6558444.

[33] ScaleFlux Computational Storage. URL `https://www.scaleflux.com/`.

[34] 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.

[35] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007. doi: 10.1109/TCAD.2006.884574.

[36] Amara Amara, Frédéric Amiel, and Thomas Ea. FPGA vs. ASIC for low power applications. *Microelectronics Journal*, 37(8):669–677, 2006. ISSN 0026-2692. doi: https://doi.org/10.1016/j.mejo.2005.11.003. URL `https://www.sciencedirect.com/science/article/pii/S0026269205003927`.

[37] 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.

[38] 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.

[39] SNIA. Computational storage technical working group, 2019. URL `https://www.snia.org/computational`. , Accessed: 05/03/2019.

[40] Eideticom, 2020. URL `https://www.eideticom.com/`.

[41] Pankaj Mehra. Samsung SmartSSD: Accelerating data-rich applications. In *Proc. Flash Memory Summit*, 2019.

[42] 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.

[43] NGD Systems, 2020. URL `https://www.ngdsystems.com`.

[44] Part Stock Website. Part Stock Specs for Xilinx XCZU19EG-2FFVC1760E. `http://www.part-stock.com/product-part/xilinx__XCZU19EG-2FFVC1760E.html`, 2020. [Online; accessed 21-Jan-2020].

[45] DRAM Exchange Website. DRAM Exchange. `https://www.dramexchange.com`, 2020. [Online; accessed 21-Jan-2020].

[46] Michael Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.

[47] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*. ACM, 2016.

[48] Jeff Janukowicz. How new QLC SSDs will change the storage landscape. Technical report, Micron, 5 Speen Street, Framingham, MA 01701, USA, October 2018. URL `https://www.micron.com/-/media/client/global/documents/products/white-paper/how_new_qlc_ssds_will_change_the_storage_landscape.pdf?la=en`. , Accessed: 12/7/2019.

[49] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale AI applications. 16(4), oct 2020. ISSN 1553-3077. doi: 10.1145/3415580. URL https://doi.org/10.1145/3415580.

[50] Mahdi Torabzadehkashi. *SoC-Based In-Storage Processing: Bringing Flexibility and Efficiency to Near-Data Processing*. University of California, Irvine, 2019.

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

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

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

[54] 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.

[55] Guest Author. What is EDSFF form factor?, Oct 2019. URL https://www.storagereview.com/what-is-edsff-form-factor.

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

[57] 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.

[58] Mahdi Torabzadehkashi, Siavash Rezaei, Ali HeydariGorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. Computational storage: an efficient and scalable platform for big data and HPC applications. *Journal of Big Data*, 6(1):1–29, 2019.

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

[60] Mahdi Torabzadehkashi, Ali Heydarigorji, Siavash Rezaei, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. Accelerating HPC applications using computational storage devices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1878–1885. IEEE, 2019.

[61] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. HyperTune: Dynamic hyperparameter tuning for efficient distribution of DNN training over heterogeneous systems. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–8, 2020.

[62] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. Stannis: Low-power acceleration of DNN training using computational storage devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi: 10.1109/DAC18072.2020.9218687.

[63] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. Stannis: Low-power acceleration of DNN training using computational storage devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, jul 2020. doi: 10.1109/dac18072.2020.9218687. URL `https://doi.org/10.1109%2Fdac18072.2020.9218687`.

[64] Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI*, pages 265–283, 2016.

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library, 2019.

[66] The Theano Development Team, Rami Al-Rfou, et al. Theano: A Python framework for fast computation of mathematical expressions, 2016.

[67] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[68] Reza Asadi and Amelia C Regan. A spatio-temporal decomposition based deep neural network for time series forecasting. *Applied Soft Computing*, 87:105963, 2020.

[69] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[71] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[72] Riccardo Miotto, Li Li, Brian A. Kidd, and Joel T. Dudley. Deep patient: An unsupervised representation to predict the future of patients from the electronic health records. *Scientific Reports*, 6, 2016. URL `https://api.semanticscholar.org/CorpusID:4404566`.

[73] Alan Ritter, Colin Cherry, and William B. Dolan. Data-driven response generation in social media. In Regina Barzilay and Mark Johnson, editors, *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 583–593, Edinburgh, Scotland, UK., July 2011. Association for Computational Linguistics. URL https://aclanthology.org/D11-1054.

[74] Yu Ji, YouHui Zhang, WenGuang Chen, and Yuan Xie. Bridging the gap between neural networks and neuromorphic hardware with a neural network compiler, 2018.

[75] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, jul 2011. ISSN 0001-0782. doi: 10.1145/1965724.1965751. URL https://doi.org/10.1145/1965724.1965751.

[76] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8, 2009. URL https://api.semanticscholar.org/CorpusID:261081944.

[77] How many Google searches per day?, May 2020. URL https://serpwatch.io/blog/how-many-google-searches-per-day/.

[78] Rachel Meltzer. How Netflix uses machine learning and algorithms. URL https://www.lighthouselabs.ca/en/blog/how-netflix-uses-data-to-optimize-their-product.

[79] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[80] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. High-performance, distributed training of large-scale deep learning recommendation models, 2021.

[81] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. The architectural implications of Facebook's DNN-based personalized recommendation, 2020.

[82] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H Chou. In-storage processing of I/O intensive applications on computational storage drives. *arXiv preprint arXiv:2112.12415*, 2021.

[83] 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, 2019. doi: 10.1109/EMPDP.2019.8671589.

[84] Cornelia Győrödi, Robert Győrödi, George Pecherle, and Andrada Olah. A comparative study: MongoDB vs. MySQL. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–6. IEEE, 2015.

[85] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. MongoDB vs. Oracle–database comparison. In *2012 third international conference on emerging intelligent data and web technologies*, pages 330–335. IEEE, 2012.

[86] VOSK offline speech recognition API. URL `https://alphacephei.com/vosk/`.

[87] Keith Ito and Linda Johnson. The LJ speech dataset. `https://keithito.com/LJ-Speech-Dataset/`, 2017.

[88] James Ng. Content-based recommender using natural language processing (NLP), Apr 2020. URL `https://towardsdatascience.com/content-based-recommender-using-natural-language-processing-nlp-159d0925a649`.

[89] GroupLens Research. MovieLens, Mar 2021. URL `https://grouplens.org/datasets/movielens/`.

[90] DigitalOcean. How to perform sentiment analysis in Python 3 using the natural language toolkit (NLTK), Jan 2021. URL `https://www.digitalocean.com/community/tutorials/how-to-perform-sentiment-analysis-in-python-3-using-the-natural-language-toolkit-nltk`.

[91] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *Processing*, 150, 01 2009.

[92] Brian Cooper. YCSB: Yahoo! cloud serving benchmark. URL `https://github.com/brianfrankcooper/YCSB`.