UNIVERSITY OF CALIFORNIA

Los Angeles

Inter-Device Communication in Near Storage Computation

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Mrunal Patel

2022

ABSTRACT OF THE THESIS

Inter-Device Communication in Near Storage Computation

by

Mrunal Patel
Master of Science in Computer Science
University of California, Los Angeles, 2022
Professor Jason Cong, Chair

Near storage computation has increasingly become a focus in improving the performance of big data systems. Technological trends have moved the bottleneck of data intensive workloads to the interconnects used to move data from storage to memory. This has given a rise to the need for moving processing power closer to where the data is stored. The solution presented in this paper aims to provide a developer friendly approach to computational storage that allows multiple computational storage capable devices to be used effectively by enabling data transfer between computational storage devices directly. In this work, we build a model system on Amazon AWS and run a merge sort workload to evaluate the benefits of allowing device to device communication. We identify the scenarios in which device to device communication is effective and propose additional optimizations and improvements to better the overall solution.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Professor Jason Cong, without whom none of this would have been possible. Thank you for your patience, mentorship and teachings. I would also like to thank the rest of my thesis committee: Professor Harry Xu and Professor Tony Nowatzki. Their encouragement and advice was priceless. A special thank you to Michael Mesnier and his team at Intel Labs, for their feedback and willingness to collaborate by sharing their system as a baseline.

I have a special appreciation for my lab mates: Daniel Tan, Karl Marett, Atefeh Sohrabizadeh, Jie Wang, Weikang Qiao, Jason Lau, Yuze Chi, Neha Prakriya, and Michael Lo. While our time together was cut short by external circumstances, I have no doubt I will cherish these friendships forever. Thank you all for being supportive and friendly throughout my time at UCLA.

I would like to emphatically thank the staff at UCLA. First, Alexandra Luong, thank you very much for coordinating everything bureaucratic in the lab - from funding to ordering equipment to arranging events and maintaining our lab spaces - we are lucky to have you around to ask for help. Second, I would like to thank the Graduate Student Affairs Officers: Juliana Alvarez and Joseph Brown. Thank you for being extremely patient and responsive in helping me achieve this goal. It's safe to say that I did not have a traditional path to the finish line and your guidance, patience and help is the primary reason any of this is possible. Thank you for your support.

I would like to thank Rob Roy and Professor Jennifer Wong-Ma for being extremely supportive and encouraging when I needed it the most. And, last but not the least, I would like to thank my parents, Anjana Patel and Kaushik Patel, and my wife, Risa Shirai, for having an unwavering faith and pushing me even when I didn't believe in myself.

Lastly, thank you to SRC CRISP for supporting this work through grant GI18518.156870.

CHAPTER 1

Introduction

Computational storage or near storage computation (NSC) systems are becoming increasingly popular as the amount of data being processed vastly increases. Another technological trend that is driving the need for NSC systems is the fact that storage bandwidth growth is significantly outpacing the peripheral connection bandwidth available for CPUs. Amazon Web Services recently announced an upgrade to their data warehousing product, AQUA for Redshift, that uses computational storage as a solution to the ever growing demands of data processing pipelines. Samsung has also recently announced the SmartSSD to create a solution that can efficiently utilize computational storage. Computational storage has been studied fairly extensively, however, there is a distinct lack of a programmer friendly scalable NSC systems.

This body of work will first provide a literature review of the present Near Storage Computation systems in Chapter 2 - studying their novelty and contribution to the field. Chapter 2.1 will first go over the origins of computation storage. Chapter 2.2 will cover general purpose computational storage systems. Chapter 2.3 will cover specialized storage systems that aim to improve the performance of specific applications. All papers are described in chronological order within their respective sections.

The work also identifies key design aspects of computational storage systems: 1. ease of use, 2. scalability and finally 3. bandwidth efficiency. We will then proceed to describe an effort made to better understand the the scalability of NSC by presenting the **Sca**lable **N**ear Storage **Com**putation **S**ystem (SCANCOMS) system that enables us to explore how

Table 1.1: Summary of acronyms used in this work.

Acronym	**Full Form**	**Brief Description**
NSC	Near Storage Computation	The idea of performing computation near where the data is stored (focused on persistent storage).
SCANCOMS	Scalable Near Storage Computation System	Key contribution of this work. Proposes a system that allows NSC devices to communicate with each other.
NESCAFE	Near Storage Compute Aware Filesystem	A component of SCANCOMS, it is a filesystem that is aware of computational capabilities of the storage devices.
COSTA	Computation Storage Acceleration Orchestrator	A component of SCANCOMS, it allows the NSC devices to communicate with each other and provides some synchronization capabilities.
INSTANT	In-Storage Networked Testbed	An implementation of SCANCOMS. It is based on an extension to the NVMe protocol implemented by Intel.

multiple computational storage devices can effectively utilized. SCANCOMS is designed to be easy to use and effectively reap the benefits of computational storage. Chapter 3 provides a description of the architecture of our novel system that allows programmers to easily and effectively reap the benefits of computational storage. We also detail our implementation, **In-St**or**a**ge **N**etworked **T**estbed (INSTANT) of the system in Chapter 4 followed by an evaluation of the system in Chapter 5. Finally, we provide a summary of our learnings and explore future improvements to the system that can enhance our work in Chapter 6.

The work contains many acronyms so we have provided a small summary of the commonly used acronyms and their associated meanings. These will be further elaborated in Chapter 3 and Chapter 4.

CHAPTER 2

Literature Review

2.1 History

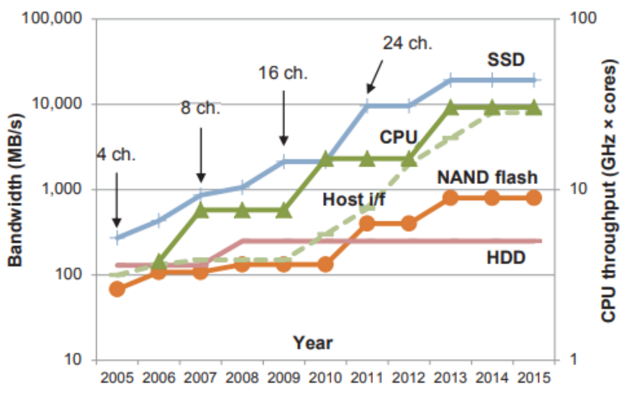

Figure 2.1: Bandwidth trends for the HDD, flash memory, SSD, host interface and CPU from. Figure produced by Cho et al. [CPO13]

The notion of bringing computational capabilities to storage devices dates back to the late 1990s. It is around this time that researchers foresaw the end of Dennard scaling and noticed trends that storage densities were increasing faster than processor performance and that processors and storage were increasingly becoming more affordable [AUS98]. Researchers began exploring computational storage as a viable solution to alleviate the performance gap between storage and processing.

2.1.1 Active disks: programming model, algorithms and evaluation [AUS98]

This work by Acharya et al. proposes adding significant processing power and memory to allow application specific code to be run on the data that is being read from/written to the disk. Their approach uses a stream-based programming model, where a disklet, a process running on the disk, can only work on data that is requested by the host. By not allowing disklets to initiate any sort of I/O, the requirements from a disk itself are simplified from a security and filesystem standpoint. The disks run a specialized operating system that manages memory and the scheduling of these disklets.

The workloads covered by the paper include SQL queries such as selection and grouping, sorting, image convolution and generating composite images. These workloads are still extremely relevant today and can be applied to several applications today. Not all of these workloads emphasize data reduction, as the goal of this paper was to scale computational capability along with storage space. They study the impact of the active disks in terms of, both, execution time and bandwidth requirements, demonstrating that even if there is no data reduction, it is possible to have overall execution speedup. These results, however, were obtained using a simulation model.

2.1.2 A case for intelligent disks (IDISKs) [KPH98]

This paper explores potential implementations of active disks, explores the various approaches to the problem of dealing with large storage systems, specifically in the context of database systems, and provides insights into research challenges and direction. They explore alternatives to fully powered servers to enable storage of large amounts of data while also scaling computational capability. Keeton et al. propose various hardware and companion software architectures for the intelligent disk system. They suggest a system where each disk runs a reduced operating system and is capable of performing a subset of database operations such as scan, sort and join on disk. The disks will be connected serially and as such can be connected in any topology from point-to-point rings to fully interconnected switches.

Table 2.1: Summary comparing features of different general purpose computational storage systems.

Related Work	**NSC Processor**	**Direct IO**	**Channel parallelism**	**Multi Disk**	**Multi Tenant**	**File System**	**Programming Model**	**Host Language**	**Device Language**
ActiveFlash	CPU+CGRA	✗	✗	✗	✓	EXT4	MapReduce	C/C++	C/C++ (limited)
Willow	CPU	✓	✗	✗	✓	EXT4	RPC	C/C++	C/C++ (limited)
BlueDBM	FPGA	✓	✓	✓	✗	RFS/EXT4	Streaming	C/C++	RTL
Biscuit	CPU	✓	✗	✗	✓	EXT4	Streaming	C/C++	C/C++
Morpheus	CPU	✓	✗	✗	✓	EXT4	Streaming	C/C++	C/C++ (limited)
Summarizer	CPU	✓	✗	✗	✓	EXT4	Streaming	C/C++	C/C++
Astoria	CPU	✓	✗	✓	✓	EXT4	Streaming	C/C++	C/C++
INSIDER	FPGA	✓	✓	✗	✓	EXT4	Streaming (one pass)	C/C++	C/C++ (HLS)
MetalFS	FPGA	✓	✓	✗	✗	FUSE	Streaming (one pass)	C/C++	C/C++ (HLS)

2.2 General Purpose Computational Storage Systems

The benefits of reducing data movement (saves power, reduces latency) and utilizing the higher available throughput of modern storage devices (improves performance), can be realized by computational storage systems. However, in order to increase the adoption of computational storage as a viable approach for general applications, it needs to be democratized. Improving the ease of user programmability of the systems and the flexibility are key steps towards improving the viability of computation storage systems. As such, researchers have also deeply explored the various implementations of more general purpose computational storage systems, exploring different types of hardware, device designs, and programming models.

2.2.1 Active disk meets flash: a case for intelligent SSDs [CPO13]

This work from 2013 describes the need to reconsider the requirements for processing near storage in the context of modern storage devices. SSDs have much higher bandwidth than traditional spinning disks and have a lot of internal parallelism available to exploit as each flash channel can be read independently. As such they propose augmenting SSDs to also contain parallelized data stream processors, akin to course grained reprogrammable arrays (CGRAs), as part of the flash micro controllers (FMCs). The rest of the hardware design is rather unchanged compared to a regular SSD - the device still retains the embedded CPU and DRAM that regular SSDs have. The software model for this was mapped to the MapReduce [DG08] frame work where the Map phase is done by the reprogrammable FMCs and the reduce phase is performed by the embedded CPU. However, the input data can only be sourced from the flash and not directly from the host - allowing processing to only happen during reads.

Cho et al. explain that the data intensive workloads will be benefited by such a device. They profile workloads to find workloads that have a high number of cycles per byte, a product of instructions per byte and cycles per instruction. They test kernels such as linear regression, string match, k-means and a scan. They find that exploiting the internal bandwidth and parallelism available in an SSD is an effective approach for workloads with high cycles per byte. They also find that workloads that have high cycles per byte due to computational intensity are benefited more than other workloads, which might be a side effect of the type of accelerator they use in storage.

2.2.2 Willow: a user-programmable SSD [SGB14]

Sudharsan et al. propose a system that effectively exposes computational storage as an RPC service. The Willow SSD is designed to have nearly identical resources as a traditional SSD. It consists of a network of computational units called Storage Processing Units (SPUs). SPUs consist of a small embedded processor, a local NVM class storage, a network interface and a programmable DMA controller. A user can program SPU applications (in C) that process chunks of data. Each SPU runs a reduced OS that provides simple multithreading, enforces file permissions, manages communication with the host-side driver, and implements protection mechanisms that allow multiple SSD apps to be active at once. The host-side software consists of a driver and a set of RPC endpoints. The application can communicate directly with SPUs through the RPC mechanism.

The well defined programming interface provided by Willow makes it straightforward to implement various applications using the new capabilities of the SSD. They have various case studies to demonstrate the flexibility of their system. They implement SSD applications that can support features such as direct IO access for applications, atomic writes, disk caching, key-value store acceleration, offload of file operations (append). While the work allows users to implement SSD support for a variety of tasks due to the flexible and generic programming model, it does not exploit the parallelism available within the SSD very effectively.

2.2.3 BlueDBM: an appliance for big data analytics [JLL15]

BlueDBM is an example of a full hardware/software system that uses NSC to efficiently process large volumes of data. At the heart of the system is a custom board that consists of an FPGA with on-board DRAM, a multi-ported network interface, 1TB of flash storage and a PCIe host interface. The flash storage is managed directly by the host software to minimize access latencies and the system uses a global addressing system such that a device can directly query data from another device. The system is networked using serial connections and has no need for a router or switch as the devices can support packet forwarding, allowing BlueDBM to be arranged in any topology desired. The system allows flash to be accessed in three different ways: directly from the host, directly from the NSC accelerator, and over the network. The software interface allows user applications to directly access the NSC accelerator or flash. They request the physical mapping of files from the filesystem and forward the addresses returned to the BlueDBM device.

To evaluate the system, excluding microbenchmarks for latencies and bandwidths, Sang Woo et al. implement some core kernels that are commonly used by big data analytics systems: nearest neighbor search, graph traversal and string search. They find that processing near storage can be very beneficial, even with little to no data reduction, when the workload is IO bound and is limited by the rate at which the raw data can be transferred. It is also effective in freeing up CPU cycles for other work. They also find that direct communication between devices can lead to noticeably improved performance as there is no software intervention to access remote data.

BlueDBM has been used as a baseline to implement several different big data systems [XLJ16, JXA17, JWZ18], with AQUOMAN [XBH20] as recent as 2020. It is a proven computational storage system that scales efficiently due to its interconnected storage devices and efficient software stack. BlueDBM, however, does not provide much insight for multi-tenant utilization of the computational storage system and does not use any standard protocols for communication. This could make it unappealing to users to move their applications to BlueDBM.

2.2.4 Biscuit: A Framework for Near-Data Processing of Big Data Workloads [GYB16]

Biscuit's primary focus is on the software and programmability aspect of NSC systems. They don't propose a specialized SSD but rather, they simply update SSD firmware to support the Biscuit runtime. The notion is to use a programming model that is familiar to also enable NSC. They provide a system that allows C/C++ code to be run on the SSD, abstracts application-device communication, efficiently uses the resources available on the SSD and guarantees safety and data integrity. The idea is to divide up the program into tasks and assign some of these tasks to the SSD (called SSDlets). SSDlets have a simple programming model, they read data from input 'ports' and output data to output 'ports'. As such, they can be chained and communication between host and device can be abstracted away from the programmer. The dataflow (i.e. how the SSDlets are chained) is specified by the host program. SSDlets can dynamically allocate memory and access files regularly (they don't have access to block level information for security purposes) as features provided by the Biscuit runtime managing the SSDlets. Since SSDlets are compiled to binaries, they can also be dynamically loaded as required.

The evaluation of the design demonstrates applications such as graph traversal, string matching, DB scan and filtering. For workloads like graph traversal, the lower latency access to data makes NSC an attractive option (as shown by BlueDBM as well). There is also significant energy saving and performance improvement when the overall amount of data transferred is reduced and the higher internal bandwidth is used optimally. Biscuit provides a very clear and intuitive framework, it also maps well to the MapReduce framework and each phase of the MapReduce framework can easily be ported to an SSDlet.

2.2.5 Morpheus: Exploring the Potential of Near-Data Processing for Creating Application Objects in Heterogeneous Computing [TZZ16]

Morpheus introduces a model for NSC systems to adhere to. They observe that object deserialization can be a major bottleneck in highly parallelized applications, especially in the context of heterogeneous computation as there can potentially be significant overhead copying the data around. To remedy this, they propose NSC where the storage device deserializes the data before transferring the data directly to the accelerator using the ability of PCIe to be able to perform a P2P transfer without the intervention of the host. By reducing the number of copies made and host intervention, the system can improve overall power efficiency, bypass OS overhead, and reduce overall traffic in the interconnect. Tseng et al. propose an extension to the NVMe protocol in the Morpheus model to support the initialization and deinitialization of a computational storage kernel, and special read and write commands to execute the the kernel on some data. They also provide a driver that supports P2P communication between the NSC device and accelerators. The programming model dictates that SSD applications work on an incoming stream of data and can be written using a limited feature set of C/C++. Their experimental results show that the Morpheus SSD can improve the performance of the overall heterogeneous system by performing deserialization prior to execution. Only some workloads they tested benefitted from the reduced data copy enabled by the P2P transfers, these kernels were IO bound and not compute bound.

2.2.6 Summarizer: trading communication with computing near storage [KMI17]

Koo et al. propose a computational storage system that predominantly focuses on read operations. Since many database operations consist of queries that are used to analyze data, read operations dominate workloads for many big data analytics systems. They propose a system by slightly modifying the SSD controller to support a task queue structure, a task controller module and a user function stack. They propose extending the existing NVMe protocol to add a few new commands to support computational storage. A task is first initialized with a tag. During this initialization step the task's local variables and any temporary data are also initialized. Then a special read request tagged with the id of the desired task is sent to the SSD controller. The controller notices the special request and adds it to the task queue with the appropriate tag and a read is performed. When the data has been read from the flash and put into the SSD DRAM, the controller checks the compute bit in the task queue and if it is set, provides the read data as input to the task. The task is executed on the CPU and the results are sent to the host upon completion.

The evaluation of the design uses SQL queries to benchmark performance and also studies the impact of internal to external bandwidth ratios and the impact of improving the computational power of the SSD. They conclude that that improving the computational power of the SSD has a much greater impact on the overall performance of the system that simply relying on bandwidth difference. This highlights the shortcoming of using low powered CPUs as the unit of processing for near storage computation.

2.2.7 Respecting the block interface – computational storage using virtual objects [AKM19]

In this paper, researchers at Intel describe Astoria, a computational storage system that extends existing block level protocols to include computational storage commands. By extending existing block level storage protocols, such as iSCSI and NVMe-oF, computational storage commands can be forwarded to storage devices without significant software changes. This also permits applications to use computational storage with minimal changes to the application itself. They use a notion of virtual objects which are essentially lists of block address and data length tuples. A virtual object can be composed of multiple independent blocks and multiple virtual objects can be worked on as part of one compute descriptor. They rely on the existing filesystem to enforce file permission and also get the block addresses to build a virtual object. Once a compute descriptor is sent to the storage device, the computation that was requested is carried out on the data specified by the virtual object. Astoria supports both read and write type operations.

The system is implemented on a remote JBOF server that is exposed as a block device to the host system. In their evaluation, they demonstrate that the system is capable of performing offloaded operation at a higher throughput than if all the data for retrieved from the JBOF to the host first. They explore other workloads such as sorting, image processing for machine learning, and LSM-tree compactions as well.

2.2.8 INSIDER: Designing in-storage computing system for emerging high-performance drive [RHC19b]

INSIDER is a state of the art near storage computation system. It was created by Zhengyuan Ruan under the supervision of Prof Jason Cong at UCLA. The system proposes using FPGAs as the medium of computation for near storage accelerators. The rationale is that a major shortcoming of using embedded CPUs is that they are not powerful enough to fully utilize the internal bandwidth, this can be observed in the results of [KMI17]. FPGAs can efficiently exploit the full extent of the internal bandwidth due to the inherent parallelism of the hardware.

INSIDER has three distinct portions. First, the user application library is designed to be simple and easy to use. It is POSIX-like and exposes computational storage through virtual files. A user simply has to register an accelerator with the system and associate a virtual file to it. Similar to [AKM19], but at a different level of abstraction, virtual files are simply a list of file paths and offsets. The user can then perform reads and writes using the library provided to get the results of the computation. It should be noted that the system is capable of both read and write computations. A virtual file is converted to a list of block address and data length tuples by the INSIDER host runtime. This runtime manages permissions by querying the underlying filesystem and efficiently transfers data to and from the disk using a specialized streaming data transfer mechanism [RHL18]. Lastly, the system implements a parametrized emulated SSD. The emulated SSD is implemented on a large FPGA where a portion of the resources are dedicated to emulating flash and the rest of the FPGA houses the acceleration kernels and controller. The parametrized emulation platform allowed for an extremely accurate design space exploration of various combinations of internal and external bandwidths and studied the impact on various different workloads. Ruan et al. classify a group of workloads that are well suited for NSC; workloads that have a high input to output data ratio and are computationally intensive are ideal.

2.2.9 Accessible near-storage computing with FPGAs [SPW20]

MetalFS is another state of the art NSC system. Schmid et al. propose a system that is similar in nature to Biscuit [GYB16], however, they replace the idea of a SSDlet with an overlay architecture based on an FPGA. Their approach involves implementing various modules that support specific functionality and allowing the user application to chain them together as they see fit. The orchestration of where the inputs and outputs connect is specified by the host program, similar to [GYB16]. The data transfer to and from the device is managed by their unique software interface. They implement a custom filesystem that exposes the various accelerator modules as executables to the host system. The host system can then ”pipe” these executables together to perform the desired computation. The idea of using a regular filesystem and extending it to support computational storage is interesting and makes the system much easier to use from a programmers perspective.

2.3 Optimized Storage Systems

Researchers have explored different ways to optimize storage systems to improve performance. With the advent of modern storage technologies such as Flash storage and other types of advanced Non-Volatile Memory (NVM) storage, the disparity in the scaling of device vs CPU bandwidths was magnified, as seen in Figure 2.1. As storage devices have become faster, the latency bottleneck has shifted away from them in storage area networks (SANs).

Caulfield et al. [CS13] show that the overall latency of a SAN is increasingly determined by the software and block transport. Unlike disk based SANs, where 1.2% - 3.8% of the overall latency can be attributed to the software, the software share of the latency can be as high as 59.6% for flash based SANs and can be an astounding 98.6%-99.5% of the latency for advanced NVM storage devices. As such, computational storage has become an increasingly important focus in the 2010s.

2.3.1 QuickSAN: A SAN for Fast, Distributed, Solid State Disks [CS13]

Caulfield et al. focus on improving the latencies of modern storage network areas. They observe that modern storage devices are very low latency compared to hard disks, and that this disparity is only going to be exacerbated with the advent of NVM storage class devices (e.g. Optane, 3D NAND). This emphasizes the need for a lean software stack and efficient block transport. QuickSAN proposes an alternative to a popular SAN, iSCSI [CSM14], that is much more efficient. Since, iSCSI uses TCP/IP as the underlying transport layer, there is a huge latency overhead from the transportation of data. It should also be noted that the SCSI protocol is limited when it comes to exploiting the higher bandwidths available in modern storage devices. QuickSAN achieves the reduced latency by streamlining the software stack and allowing the SSDs themselves to connect to the network. They use a direct block interface, allowing user applications to directly access block storage without going through the slow operating system IO stack. The implementation requires a lossless network to run and this is helpful in avoiding the expensive TCP/IP stack.

Modern industry standard SANs are inspired by similar ideas. Non Volatile Memory express over Fabrics (NVMeoF) [Wor19b] is a modern storage area network protocol that has a very lean processing stack, unlike iSCSI. They also initially only supported lossless networks but have recently expanded to support TCP/IP as an underlying transport as well.

2.3.2 An efficient design and implementation of LSM-tree based key-value store on open-channel SSD [WSJ14]

Wang et al. describe a technique they used to improve the performance of LSM-tree based key-value stores, specifically LevelDB [DG17]. They build their system on top of a special SSD device built by Baidu's researchers, SDF [OLJ14]. The SDF is an open-channel (i.e. exposes all the flash channels to software) SSD built by connecting a large FPGA to 4 smaller FPGAs, each of these smaller FPGAs is connected to 11 flash channels. The software interface for the SDF device exposes each of the channels as an independent device allowing software applications to fully control the parallelism available in the device. The standard implementation of LevelDB dumps a single immutable memtable, the data in-memory data structure that keeps track of the keys and associated values, when a size threshold is reached. Wang et al. increase the number of immutable memtables to match the number of channels exposed by SDF to allow for maximum concurrency during the writes, resulting in a significant improvement in the throughput of the system.

The paper motivates the significance of having a high throughput LSM-tree based KV store and its importance in modern cloud computing and large-scale distributed storage systems. They exploit parallelism that is available through their system, albeit, without using true computational storage.

2.4 Learnings

We have established that computational storage is not only a viable solution, but rather a necessary solution to address the ever growing data sizes paired with the current technological trends. We identify the main challenges of implementing a comprehensive NSC system as follows:

2.4.1 Programmability of the NSC processor

The programmability of an NSC system is of the utmost importance. This is the primary design consideration for most of the prior works studied in this paper. By making a system easy to program, the overall usability and rate of adoption is significantly improved. Researchers have explored various programming models in an attempt to identify the most suitable one. MapReduce [DG08] has been explored as a potential programming model since many big data applications already utilize it [CPO13, GYB16]. While [GYB16] use a programming model that is not limited to MapReduce, the notion of SSDlets (mini-streaming tasks) lends itself well to this model and is exemplified by the authors. The main drawback of using MapReduce is that it limits the overall flexibility of the system and many disk intensive applications (e.g. compression, encryption, etc.) don't necessarily require a MapReduce model. [SGB14] uses an RPC based approach allowing applications to run RPCs on the storage device. The main issue with this approach is that it requires applications to be ported to this model, a specialized storage device that is not only different architecturally but also uses a completely customized storage protocol that is incompatible with existing standards. While the model itself is flexible, the programming effort to use this system is prohibitively large. [AKM19, KMI17, TZZ16] propose a NSC system that extends NVMe, a modern storage protocol that is industry standard, to support computational storage commands as well. The main benefit of this is that existing applications only need minor modifications to utilize computational storage. It can be viewed as an improved implementation of the RPC based system in [SGB14] since the overhauling changes are limited to storage device and its driver. [RHC19b, SPW20] take a higher-level approach by bringing computational storage to the filesystem level. [RHC19b] utilizes the notion of virtual files (a mapping containing the block addresses from various physical files) and provides a POSIX-like interface to minimize the impact of porting applications to utilize the framework. [SPW20] implements a FUSE-based filesystem that interprets executables stored in the filesystem as in-storage applications and allowing for communication between host application and NSC application using UNIX pipes. Using a filesystem level interface makes application software easy to port and easy enforcement of access permissions and concurrent access, however, the underlying protocol used by these systems is entirely non-standard.

2.4.2 Exploiting available bandwidth efficiently and minimizing data movement

The need to reduce the amount of data being transferred around and utilize the greater internal bandwidth available in modern storage devices are the primary drivers of NSC research today. [AKM19] proposes a system that is primarily designed for moving computation closer to the data in the context of remote storage servers. Their main objective was to minimize the amount of data transferred over the network between the application server and the storage server. [TZZ16] focuses more on a single machine and the interconnect to the peripherals. They exploit the peer-to-peer nature of PCIe to allow different accelerators to request data directly from the storage system instead of relying on the host CPU to first copy the data from storage to main memory and then from main memory to the accelerator, effectively removing the unnecessary overhead of moving data to and from memory. In the context of a single machine, SSDs have a strict power budget as they are usually powered directly by the peripheral interconnect and don't have their own external power supply. As such, SSDs today have multiple "wimpy" ARM based CPUs. Most of the previous works explored various ways of utilizing these CPUs [SGB14, GYB16, KMI17, TZZ16]. [SGB14] proposes a solution that involves arranging the CPUs on a bus, each with a local flash channel, to exploit the channel-level parallelism. [GYB16] and [KMI17] don't specify whether or not the CPUs can exploit channel level parallelism but the multiple CPUs can run various tasks in parallel, like a multi-threaded system. The main issue with CPU based systems is that they are unable to effectively capture the parallelism that is available within the SSD itself, and, depending on the workload, may not be able to effectively capture the parallelism available in the computation. As observed by [RHC19a], the nature of NSC workloads lends itself well to streaming, especially if there is data reduction between the input and output. Stream processors that can efficiently exploit the available parallelism within the storage devices are better suited for NSC. This is supported by the observation made by [KMI17] which finds that the embedded CPUs on the SSD can easily be overburdened. [CPO13] attempt to extract the best of both worlds, by utilizing a course-grained reconfigurable architecture along with embedded CPUs. They process the data coming from the flash channels using the a data flow architecture than can be configured before the results are processed by the CPUs on the disk. FPGAs have also been explored as the processor of choice and with the latest improvements in FPGA design synthesis and overall flexibility of the hardware, the barrier to programming them has be greatly reduced. [JLL15, RHC19b, SPW20] use FPGAs as the NSC processor due to it's power efficiency and ability to extract the maximum parallelism available. FPGAs can also be programmed to run softCPUs should a CPU-friendly workload such as object serialization [TZZ16] be offloaded to the storage device.

2.4.3 Enabling the utilization of multiple storage devices

The diminished interest in computational storage in the 2000s can be attributed to the success of distributed storage systems. They allowed data to be stored across many machines and helped deal with the problem of scaling storage requirement effectively. Current trends (Figure 2.1) show that storage technology has been improving faster than interconnect technology, additionally, modern storage devices have much lower latency than spinning disks [CS13]. These facts have reignited the interest in computational storage once again. However, most recent work has primarily focused on programmability and processing efficiency. [JLL15] is the state of the art for NSC systems that use multiple storage devices. They build a specialized network of devices each consisting of an FPGA, flash storage, memory, and a network interface. This allows their system to communicate between storage devices directly, should the devices be configured so. The drawback of this work is that it is limited to running only one application at a time, i.e. all the storage devices work on only one task.

CHAPTER 3

Architecture

The prior works have effectively addressed at most two of the three challenges detailed in Section 2.4. In this section we will present a system that is designed to address all three challenges based on the learnings from past research. We will go over the main concepts of this system and without delving into the details of the implementation. We cover the details of a specific implementation in Section 4 and will go over some more possible implementations in Section 6. The goal of this section is to describe the solution in an implementation agnostic style as we believe this approach can be implemented in many different ways.

3.1 SCANCOMS (Scalable Near Storage Computation System)

We propose SCANCOMS (Scalable Near Storage Computation System), a computational storage system that is built around scalability, ease of use and computational efficiency. The system consists of three major components: 1) the user application API and library, 2) the block device orchestrator, and 3) the computational storage capable block device. The overall architecture of the system can be seen in Figure 3.1.

3.1.1 NESCAFE: Near Storage Compute Aware Filesystem

NESCAFE is a special filesystem that is aware of and manages computation storage and the association with files. It also provides a framework for developers define new computational storage commands. The filesystem serves to minimize the impact of porting existing

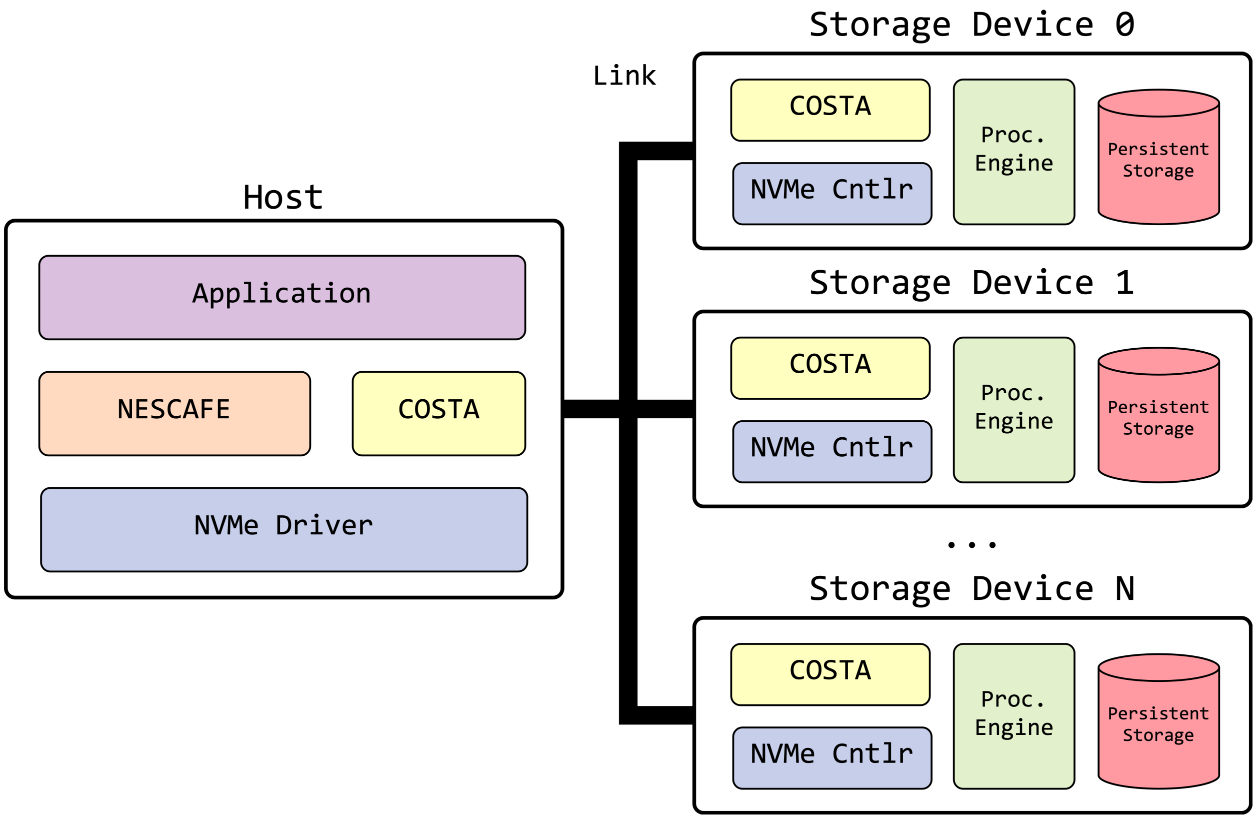

Figure 3.1: SCANCOMS architecture overview.

applications to utilize the computational storage system, inspired by [RHC19b, SPW20]. We utilize the open file descriptor table by extending the descriptor to also contain the computational storage metadata, such as the computational storage command, parameters and other configuration details. When an application requests for an IO operation on a file descriptor, the filesystem checks if there is an associated computational storage command and forwards the request to the underlying block storage. The filesystem also performs basic validity checks for computational storage requests. The primary purpose of this aspect of the system is to provide a user-friendly and familiar API to access computational storage. NESCAFE also provides a standard framework for developers who wish to implement new computational storage commands. The primary motivation behind choosing a filesystem as the layer of abstraction is due the wide-spread use of filesystems as an API for interacting with storage. We have found that the no modification to POSIX APIs are required to allow a filesystem to support computational storage as any special operations, such as registering a particular computational storage command on a file descriptor, can be performed using an `ioctl` syscall. Using a filesystem also allows for the block storage layer to be user agnostic, as is the status quo today. The burden of enforcing security and permissions can be taken on by the filesystem instead of the NSC device or block storage layer. Finally, the filesystem can provide a standard consistency model which is crucial for multi-user systems accessing the same data.

3.1.2 COSTA: Computational Storage Acceleration Orchestrator

COSTA is the part of the system that will manage and orchestrate the NSC devices. In order to modularize the system, we have separated the aspect of orchestration from the filesystem itself. This separation of concern provides the developers of the computational storage operations a more focused approach and eases the tuning of the overall system by allowing developers to use different orchestration approaches without changing other aspects of the system. COSTA is designed to work at the block level by maintaining a global block address space. This works by associating a device ID with the physical block addresses that belong to the particular device. The computational storage operations are provided these tagged block addresses as an input and, in order to receive the data, they request COSTA for data located at a specified tagged block address. COSTA then looks up where the data resides and forwards the request to the correct device or performs a local read from the persistent storage. If the read was remote, the data is transferred directly from one device to another. On the host side, COSTA will allow for orchestrating various computational storage operations and creating dependencies between them. For example, a sort command that operates on multiple disks requires at least two operations: a sort and followed by merges. The sorts have to happen first, followed by the merges. COSTA enables this by allowing blocks used by an operation to be locked and to be unlocked once the operation has processed that data. Should a different operation request data from a locked block, it will be paused until the requested block has been unlocked.

3.1.3 Computational Storage Device

The computational storage device should have the following properties: 1) It should understand a standard block layer protocol that supports computational storage. 2) It should be able to support execution of multiple operations 3) The operations should be able to receive parameters and tagged block addresses as input and be able to make requests to COSTA for the data at the specified addresses. By following this paradigm, we can make generic computational storage devices that can work in conjunction with each other regardless of hardware/software implementation details. For example, an operation can be hardware accelerated for some computational storage devices, while for others, it may be run on a CPU on the device. Both of these devices can work together as long as they follow the same block level protocol and are able to make requests to COSTA for fetching the data properly.

CHAPTER 4

Implementation

In this section, we discuss the details of our implementation of SCANCOMS. We sought to build a system where we could experiment a variety of different parameters within the SCANCOMS system to understand when disk to disk communication is most viable. We found that the work done by Intel in [AKM19] could be an important building block for our system - as their system implements an extension of the NVMe-oF block storage protocol to support computational storage. To this end, we built INSTANT (**In**-**St**or**a**ge **N**etworked **T**estbed), an implementation of SCANCOMS that allows us to tweak parameters such as the the number of computational storage devices, the link bandwidth between the devices and host, and a variety of different data sizes and operations. Our main focus in this work is on the sort workload. The algorithm we used was a merge sort, where all the sorting happens locally on the device and followed by a merge of the data across different devices. First, we will describe the baseline of this workload using Intel's OAS system [AKM19] without any modifications, followed by a description of INSTANT and how it builds on Intel's OAS system.

4.1 Intel OAS: Computational Storage using NVMe-oF

Intel OAS is a computational storage system that is based on remote block storage protocols. They have support for both iSCSI [J 04] and NVMe-oF [Wor19b] protocols. We focus on the modern NVMe-oF protocol as it is the de facto block level protocol for high performance storage devices. The main overview of the Intel OAS system can be seen in Figure 4.1. Intel

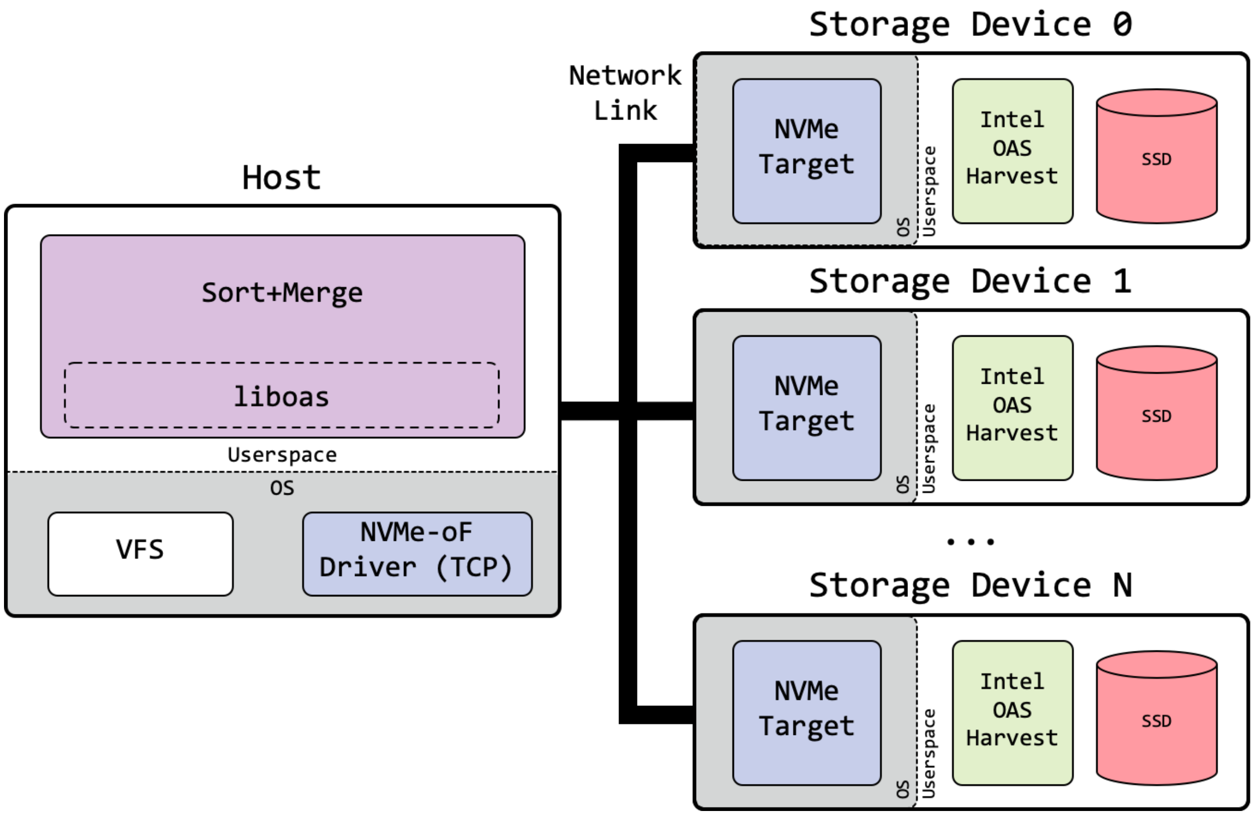

Figure 4.1: Intel OAS System Overview. The colors have been matched appropriately with the original SCANCOMS Figure 3.1. In this case, the application is a merge sort application.

OAS comprises of two major parts. The first is a library to create and trigger computational storage commands and the second is, Harvest, a process that picks up computational storage requests made by the application on the storage remote device and fulfills the request by processing the data specified. This library is utilized by the application directly. While the library takes care of certain interactions with the filesystem regarding the physical locations of data on the storage devices, the application developer still has to understand how to use the library and integrating this system into existing applications would be a hurdle. Harvest, the part of the system that receives the computational storage request, receives a series of physical block addresses and then performs a read from those addresses before invoking the requested operation on the data. If the requested data is larger than a tunable parameter (defaults to 1MB chunk), the data is read in chunks and the computational storage operation is invoked on one chunk at a time, with Harvest maintaining a customizable context between chunks. It should be noted that without any modifications Harvest can only read from the SSD that is locally attached to the server that is the NVMe-oF target (which acts as the storage device from the host's perspective).

Figure 4.2 to Figure 4.5 describe in detail how a merge sort implementation would work on a system where there are multiple NVMe-oF targets (each acting as a single disk). In this series of figures, a dotted line means the interaction occurred on the control plane and a solid line means the interaction occurred on the data plane. The purple color represents interactions that occur on the host and may involve using at most one disk at a time. The red and blue colors represent interactions that occur on the disks in parallel.

In order to perform a merge sort operation between data on multiple disks, the application needs to sort the data stored on each disk, then copy the sorted data from one disk over to the other disk and then perform a merge operation. The creation of the appropriate `oas_descriptor`s is up to the application. An `oas_descriptor` contains all the necessary metadata for the Harvest process to be able to successfully complete a request (i.e. the operation requested, the input block addresses, the output block addresses and any parameters to be sent to the computational storage operation).

Figure 4.2 visualizes the process of using the library provided by Intel to create the the sort `oas_descriptors` for each disk. Under the hood, the Intel library queries the filesystem using an FIEMAP `ioctl` to get the physical block locations of the respective files that need to be sorted. Once the descriptors have been created, the application invokes the library to trigger NVME-oF based computational storage command. This can be done in parallel for each disk. Each disk receives its respective `oas_descriptor` and the Harvest process begins processing the request. It reads data from the disk, sorts it and then finally writes it back. Once this process is completed, the NVME request is completed and control is returned to the application.

Figure 4.3 visualizes the application copying the locally sorted data from one device to another. The applications interacts with the filesystem directly in this case and performs a regular copy in order to ensure that the data can be merged remotely. Since there is no disk to disk data transfer, data is copied from device 0 to the host first and then written to device 1 from the host. This is performed because Harvest without modifications can only read from the local disks.

Figure 4.4 visualizes the application creating a new `oas_descriptor` for the merge operation. This descriptor is then sent to device 1 which completes the merge operation. Finally, a read back of the fully sorted data set is performed as seen in Figure 4.5.

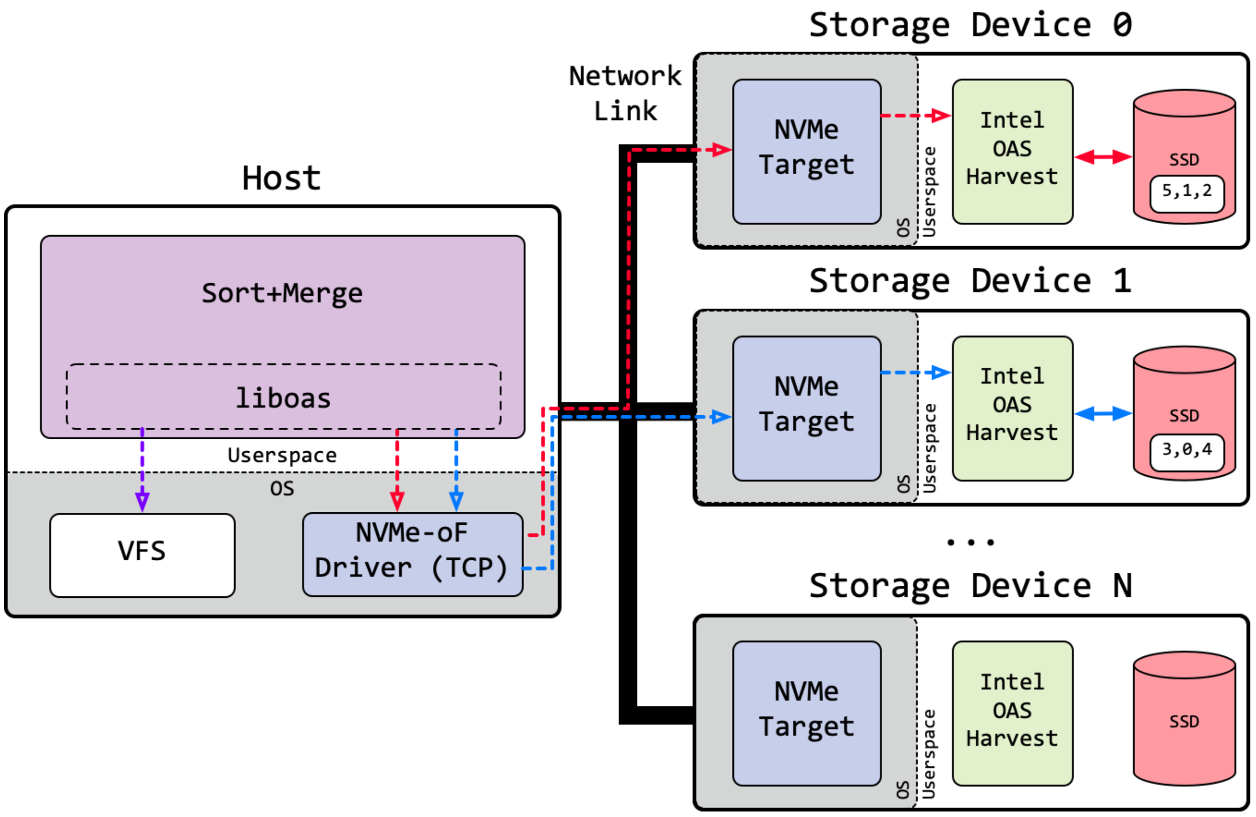

Figure 4.2: Merge sort using Intel OAS Step 1: Each disk sorts local data.

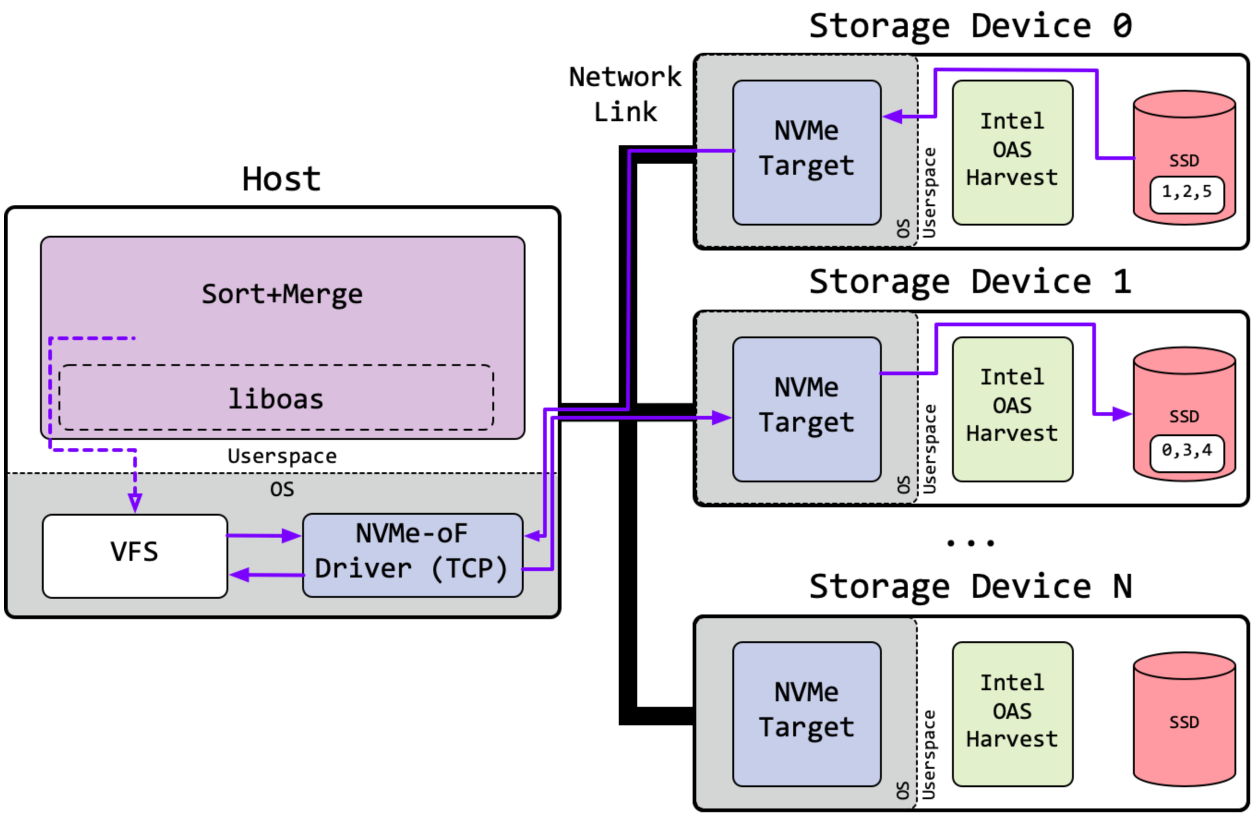

Figure 4.3: Merge sort using Intel OAS Step 2: Copy sorted data from Disk 0 to Disk 1.

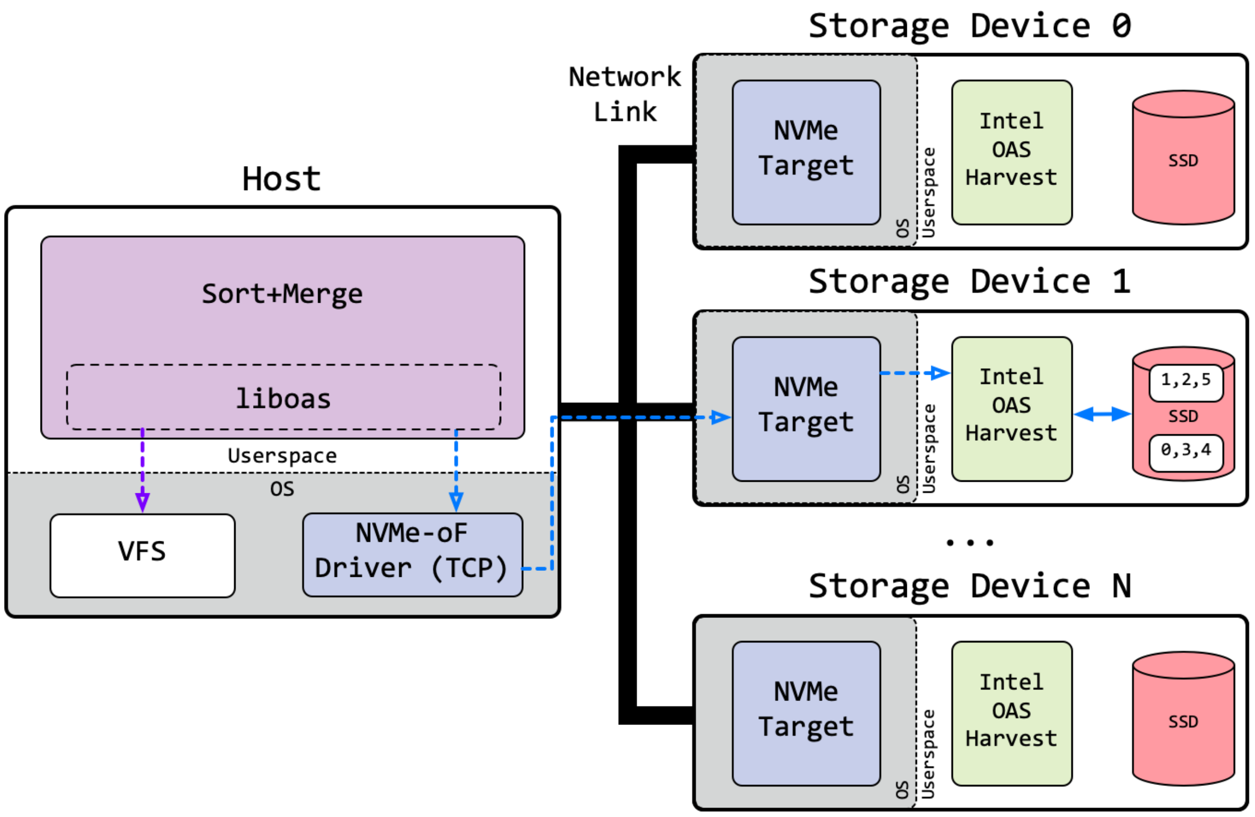

Figure 4.4: Merge sort using Intel OAS Step 3: Merge sorted data on Disk 1.

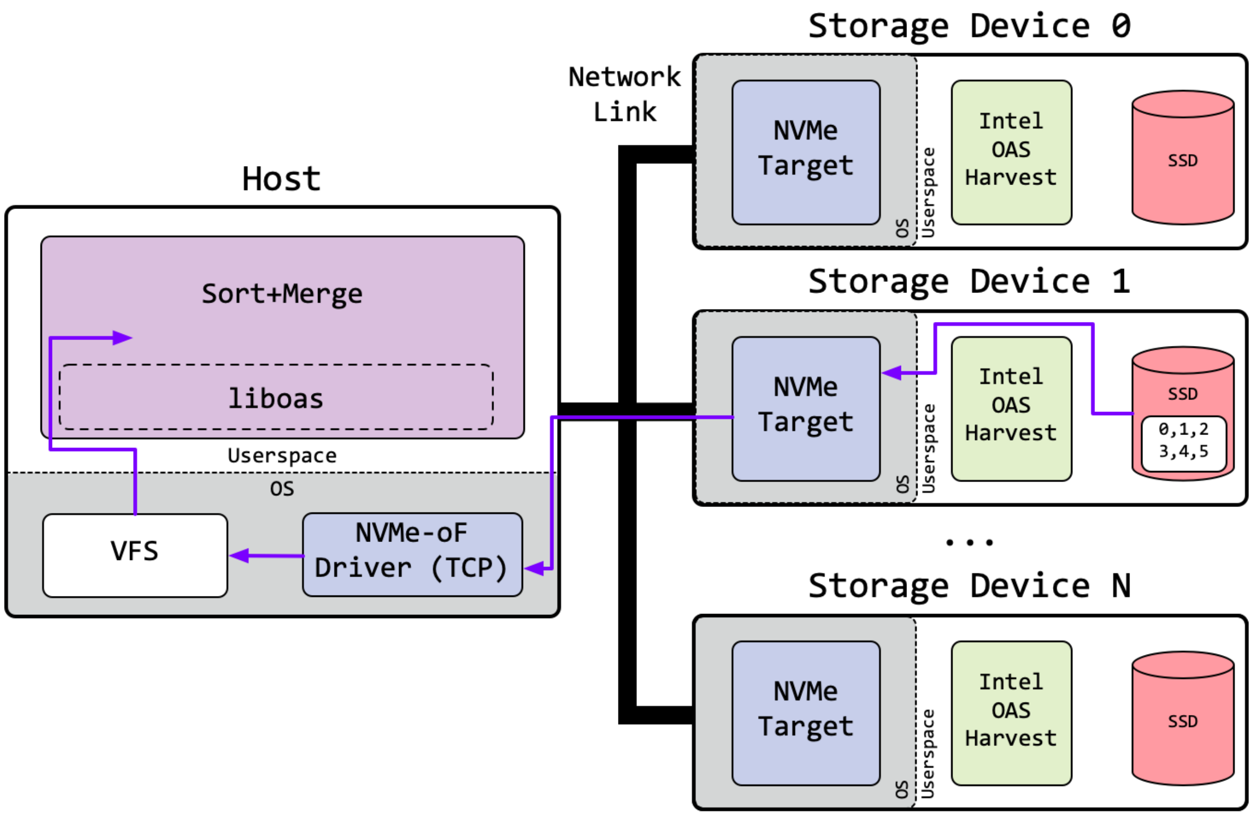

Figure 4.5: Merge sort using Intel OAS Step 4: Read sorted data back from Disk 1.

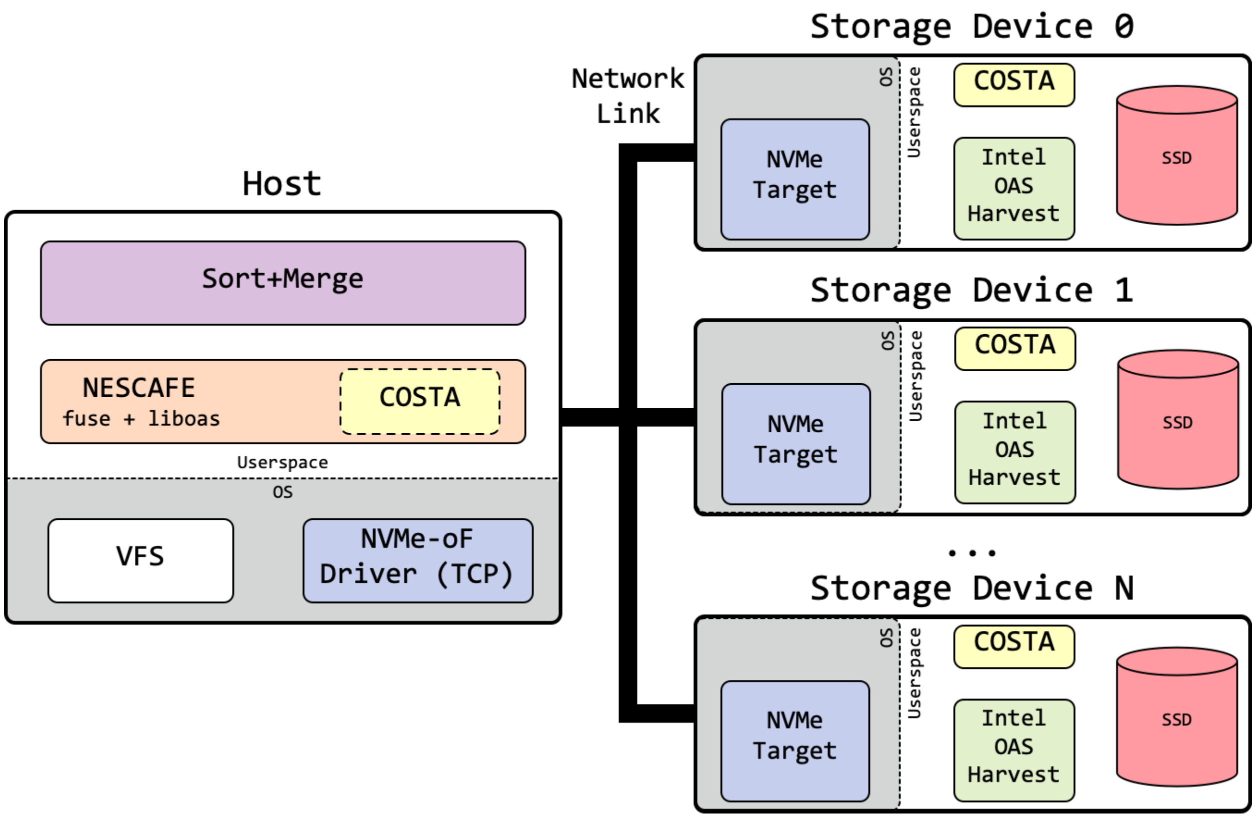

Figure 4.6: INSTANT System Overview. The colors have been matched appropriately with the original SCANCOMS Figure 3.1. As in Figure 4.1 the application is merge sort.

4.2 INSTANT: In-Storage Networked Testbed

The goal of INSTANT is to be able to support a flexible environment where parameters such as link speed between host and storage devices can be controlled, along with the number of storage devices and flexible workload sizes. We can see the overall implementation overview in Figure 4.6. In order to achieve this goal we decided to implement SCANCOMS in the cloud where some machines would act as remote storage devices that receive commands from a host machine. The idea is that we can control the network bandwidth on these machines to control different link speeds, start up more instances to add more storage devices and perform experiments on various workload sizes. There are several properties that made Intel OAS a very attractive option to base our implementation on. First, Intel OAS extended a standard block level protocol (NVMe-oF) to implement computational storage. NVMe-oF is a remote block layer protocol and can be run on TCP as well. This helps us achieve one of the important goals of INSTANT, namely, controlling the bandwidth between different components. Second, the Harvest software that processes computational storage requests can be easily modified to support tagged block addresses that are needed by COSTA in order to enable disk to disk communication. The changes to Harvest simply require replacing the standard read with an API call to the local COSTA processes. Lastly, using Intel OAS allows us to compare the effectiveness of performing disk to disk communication while comparing directly to the baseline running essentially the same core computational storage system without disk to disk communication.

COSTA is also a networked process in this implementation and as such a socket can be opened between Harvest and COSTA allowing COSTA to fulfill data requests made by Harvest. In fact, since Harvest does not directly use the block addresses provided to it except for performing the read, we take advantage of this by using the upper 9 bits of the 64 bit block address to encode the disk where the physical block resides. Since, Intel OAS is designed to work for both iSCSI (targeted towards regular HDDs) and NVMe-oF (target towards SSDs), the library defaults to 512 byte sector sizes (which is what allows us to use the upper 9 bits of block addresses).

Finally, we chose to implement NESCAFE using the FUSE library. The Filesystem in Userspace (FUSE) library provides a lot of the necessary interfaces with the Linux kernel's VFS out of the box. We only implemented the functions of a filesystem we thought were absolutely critical to allow us to create INSTANT. Additionally, to ease implementation complexity and facilitate data sharing, we integrated COSTA on the host as part of the same process as NESCAFE. Another benefit of using the FUSE library is that integration with the Intel OAS system was facilitated greatly. The Intel OAS system uses the FIEMAP `ioctl` to get information about the block addresses when creating `oas_descriptors`. By implementing a handler for this in NESCAFE, we can respond to the requests with our tagged addresses.

Figure 4.7 to Figure 4.12 describe in detail how a merge sort implementation would work on INSTANT where there are multiple computational storage devices. In this series of figures, a dotted line means the interaction occurred on the control plane and a solid line means the interaction occurred on the data plane. The purple color represents interactions that occur on the host and may involve using at most one disk at a time. The red and blue colors represent interactions that occur on the disks in parallel.

In order for for a merge sort application to offload the work to the drives in INSTANT, the application simply has to make a request to the NESCAFE filesystem. The application has to `open` the file and then use a custom `ioctl` to register the type of computational storage operation that should happen on this file. We explored options such as basing the type of operation on the file extension, however, we found that providing the user application some control over the type of computation performed was more approachable. Our implementation of NESCAFE defines a few structures that contain necessary information to be able to perform a computational storage offload. The actual computational storage operation is performed when an IO operation is actually performed on the file descriptor, which in our case would be a `read`.

Figure 4.7 shows the interaction between the application and NESCAFE in the INSTANT system. The application then performs standard file operations that are forwarded to NESCAFE by the FUSE library. NESCAFE handles all the interactions with the Intel OAS system. The merge sort operation needs to be broken down in to smaller suboperations (sorts and merges), each with its own `oas_descriptor` and this process is taken care of by NESCAFE. We have implemented a framework that allows developers to register new operations and create the appropriate descriptors required to fulfill the requirements of the operation.

Figure 4.8 shows COSTA sending lock requests to storage devices for locking the blocks that will be operated on. INSTANT implements COSTA as part of NESCAFE and provides a framework to implement different dependency chains between the operations. This is done to allow the developer to control when blocks are locked and unlocked. In the merge sort operation all the blocks are locked and then as each suboperation is completed, the blocks associated with that suboperation are unlocked.

Figure 4.9 shows the part of the process that is almost identical to that in the regular Intel OAS system. The data is sorted locally on the disks. The main difference is that Harvest has been modified to request all data from COSTA. COSTA figures out that this is a local request and fetches the data from the SSD that is local to the machine. Another difference is that the descriptor for the merge operation is also sent, however, that request is queued until device 0 can finish its sort operation first. Once the sort operations are completed, the blocks associated with the sorting are unlocked as shown in Figure 4.10

Figure 4.11 shows the disk to disk transfer that is enabled by INSTANT. Since Harvest requests for data using tagged block addresses, COSTA can figure out which device the data resides on and forwards the request to the appropriate device. This avoids a roundtrip to the host.

Finally, we perform a readback of the sorted data in Figure 4.12. The complexity of creating descriptors and moving data around is taken care of by NESCAFE in combination with COSTA, the application does not need to worry about these details at all.

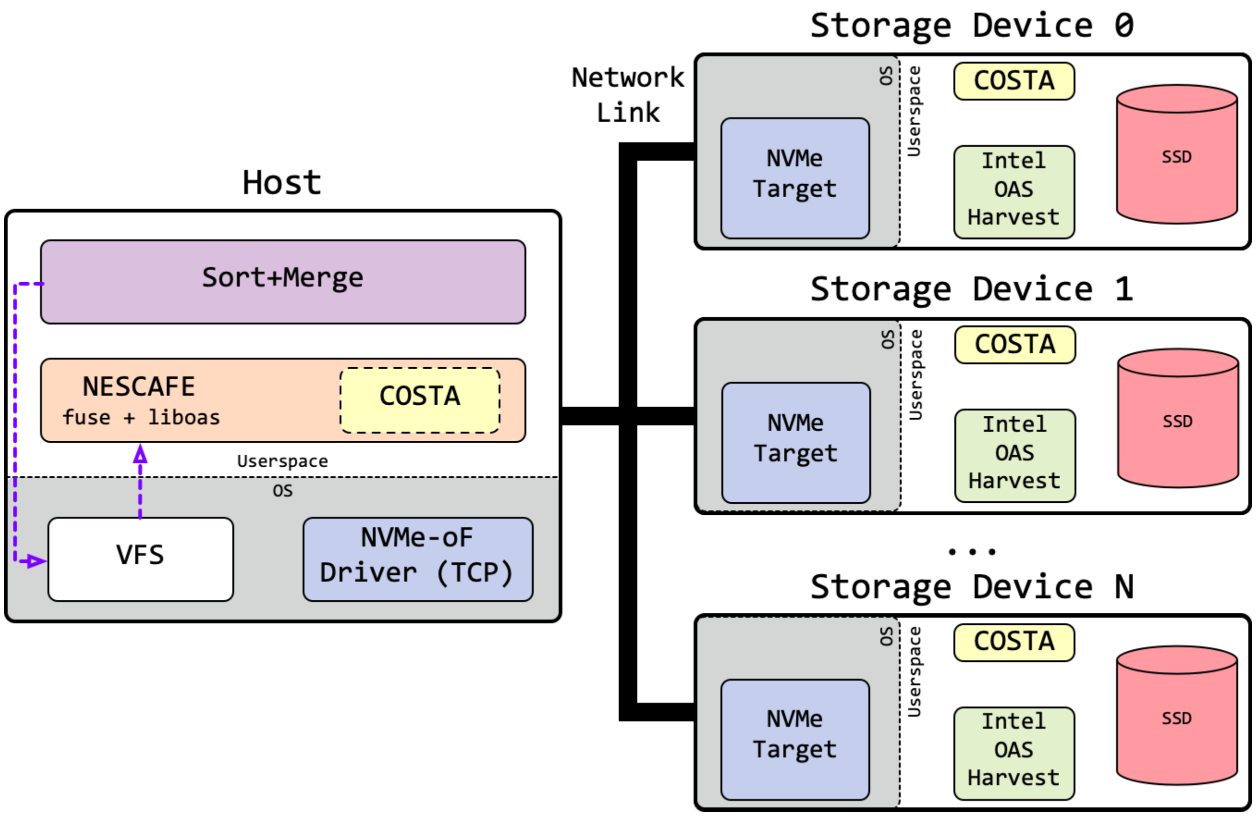

Figure 4.7: merge sort using INSTANT Step 1: Application makes request to merge sort to NESCAFE.

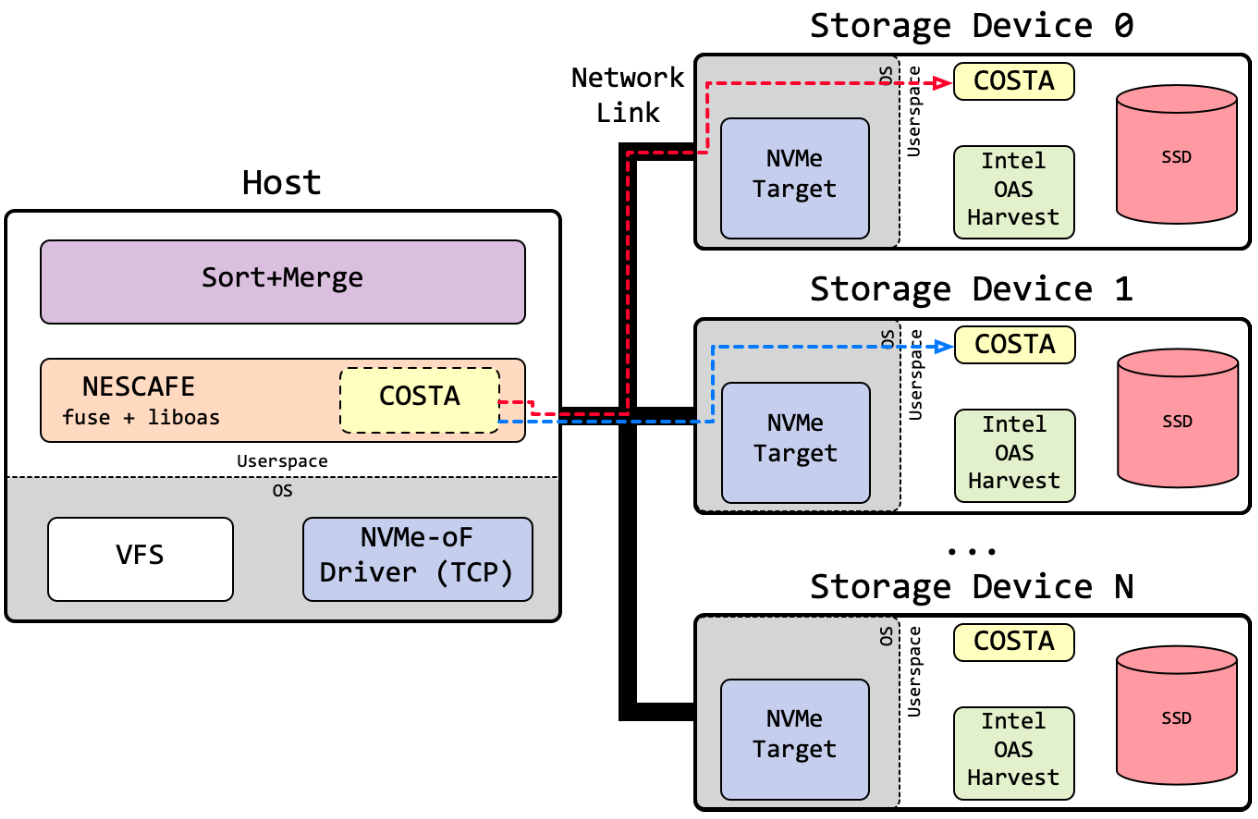

Figure 4.8: merge sort using INSTANT Step 2: COSTA locks blocks that will be operated on.

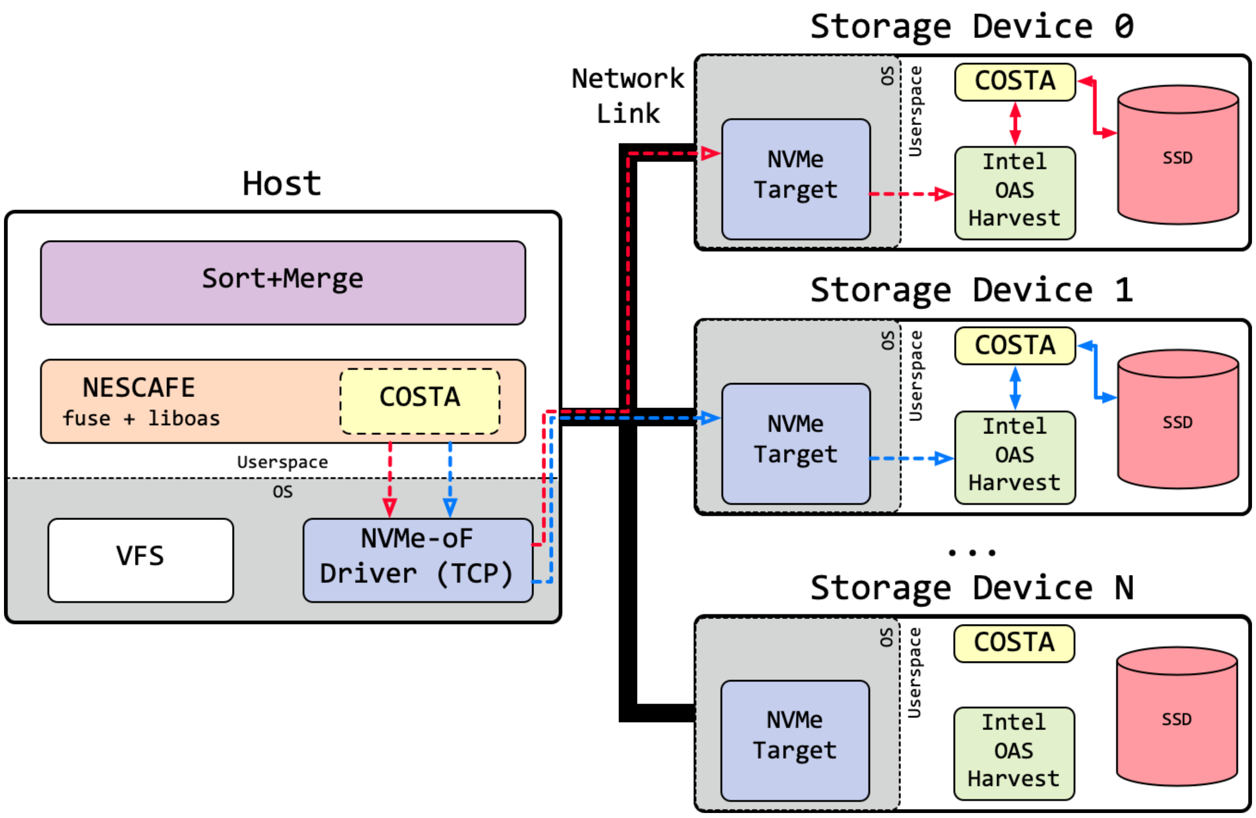

Figure 4.9: merge sort using INSTANT Step 3: Sort is performed locally on each disk.

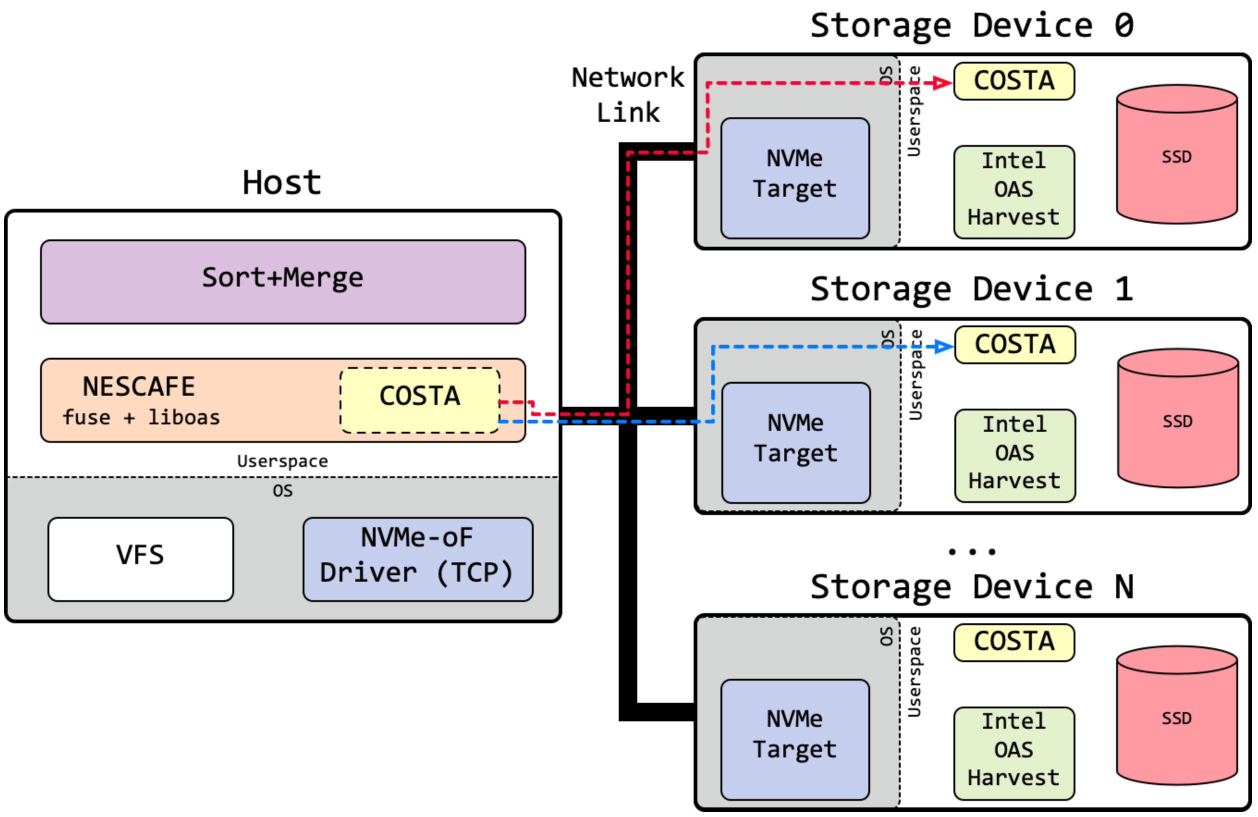

Figure 4.10: merge sort using INSTANT Step 4: COSTA unlocks blocks that have been processed.

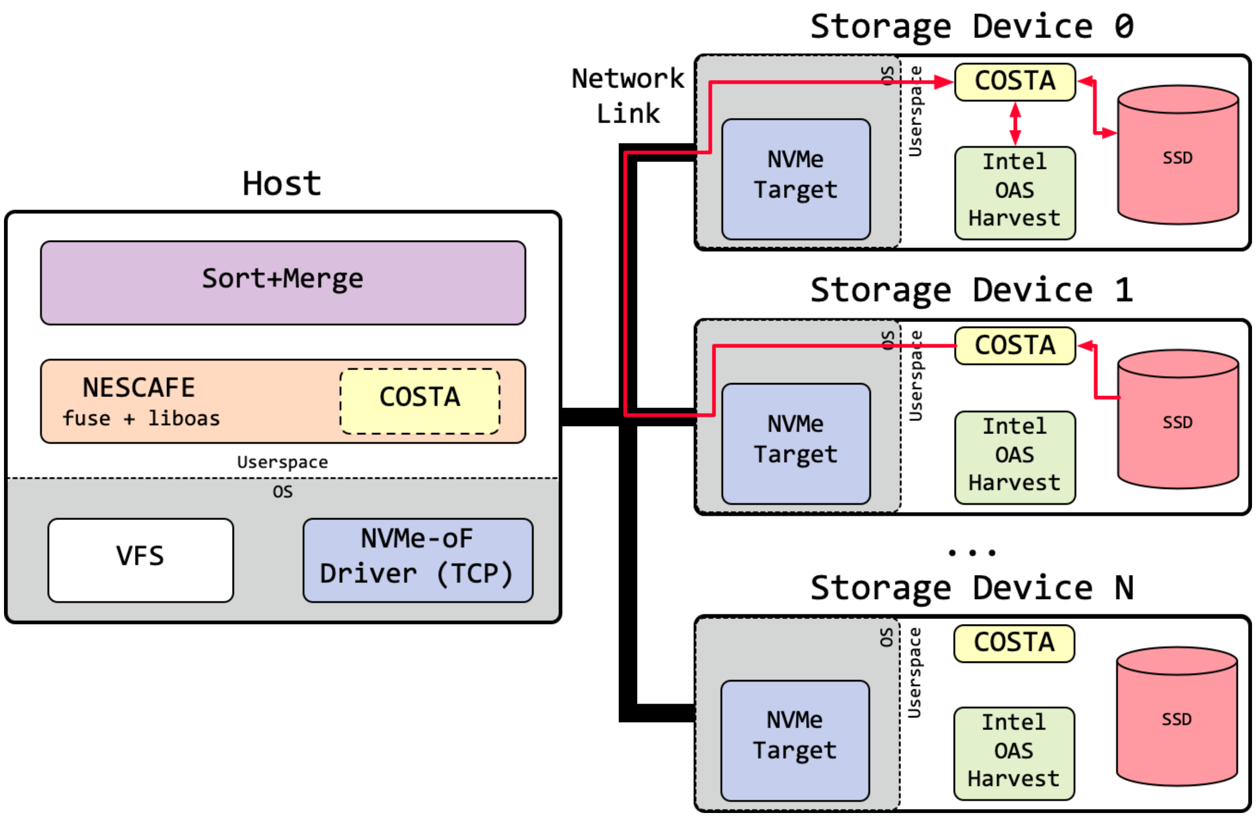

Figure 4.11: merge sort using INSTANT Step 5: Merge is performed between two disks.

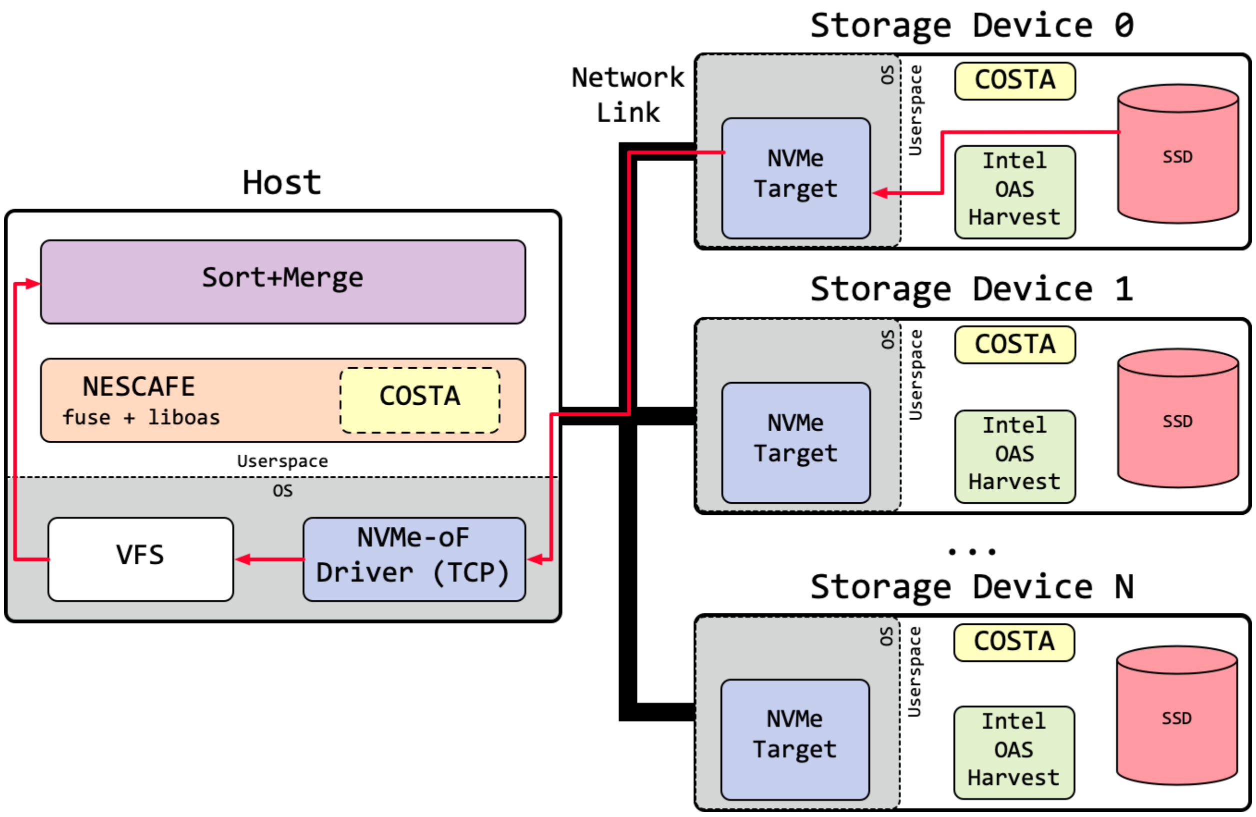

Figure 4.12: merge sort using INSTANT Step 6: Data is read back and returned to application.

CHAPTER 5

Evaluation

We evaluate the system implemented on two fronts. The first one is a qualitative evaluation for the programmability of the overall system. The second is a performance evaluation to understand better the circumstances in which a computational storage system that supports direct communication between disks is more beneficial than one that does not. In both cases we are comparing our INSTANT implementation with equivalent operations on Intel OAS. The workload we used to evaluate the system was a merge sort operation.

5.1 Programmability

As we learned in Chapter 4.1, the Intel OAS system provides a disk-side framework for developers to create new types of computational storage workloads to be offloaded to the disks. The framework, which requires the developer to implement a function that will operate on slices of the data, is very flexible and supports a wide variety of different types of workloads. We found this framework to be easy to use and sufficient in terms of programming flexibility and, as such, the INSTANT system does not alter this framework in any way and developers can create computational storage operations in the same manner as they would for the Intel OAS system.

The aspect of INSTANT that is in-charge of orchestrating the data-movement between different disks for different workloads resides within NESCAFE on the host. To allow for support of various operations we have created a flexible C++ framework that allows developers to register new types of offloads. A developer has to define a struct that contains all the parameters that can be passed to the computational storage operation, if there are no parameters, no struct definition is required. A developer can also choose to create a list of computational storage suboperations to be run for a particular type of operation. This allows users to make a request for a higher level command and not worry about the details of generating the suboperations required to achieve this. Additionally, by default the orchestration mechanism is to lock all input blocks before sequentially starting computational storage operations and unlocking them upon the completion of the respective operation. Should the developer require a more complex mechanism, they can register a callback which allows the developer to control the locking/unlocking and operation processing. For example, an offloaded sort command consists of several computational storage suboperations, i.e. sort operations and merge operations. In this case we can simply use the default orchestration mechanism as all input blocks will be locked and blocks will be unlocked as the sort and merge suboperations are completed.

We envision a world where a library of various offloads are available of users to use. From a user-application perspective, NESCAFE makes computational storage extremely easy to use. The user simply has to populate the necessary parameters for the computational storage command, register the computation storage command desired on the file and perform a standard IO operation. The added complexity to a user application is negligible and we believe this is an extremely important aspect of the system as it significantly eases adoption.

5.2 Performance Comparison

We compare the performance of INSTANT to the standard Intel OAS system. For our experimental setup we run both Intel OAS and INSTANT on the same AWS Instance Types. We use the M5DN instances on AWS as they provide large amounts of memory, high network bandwidth and finally physically connected NVMe SSD, which makes the instance type

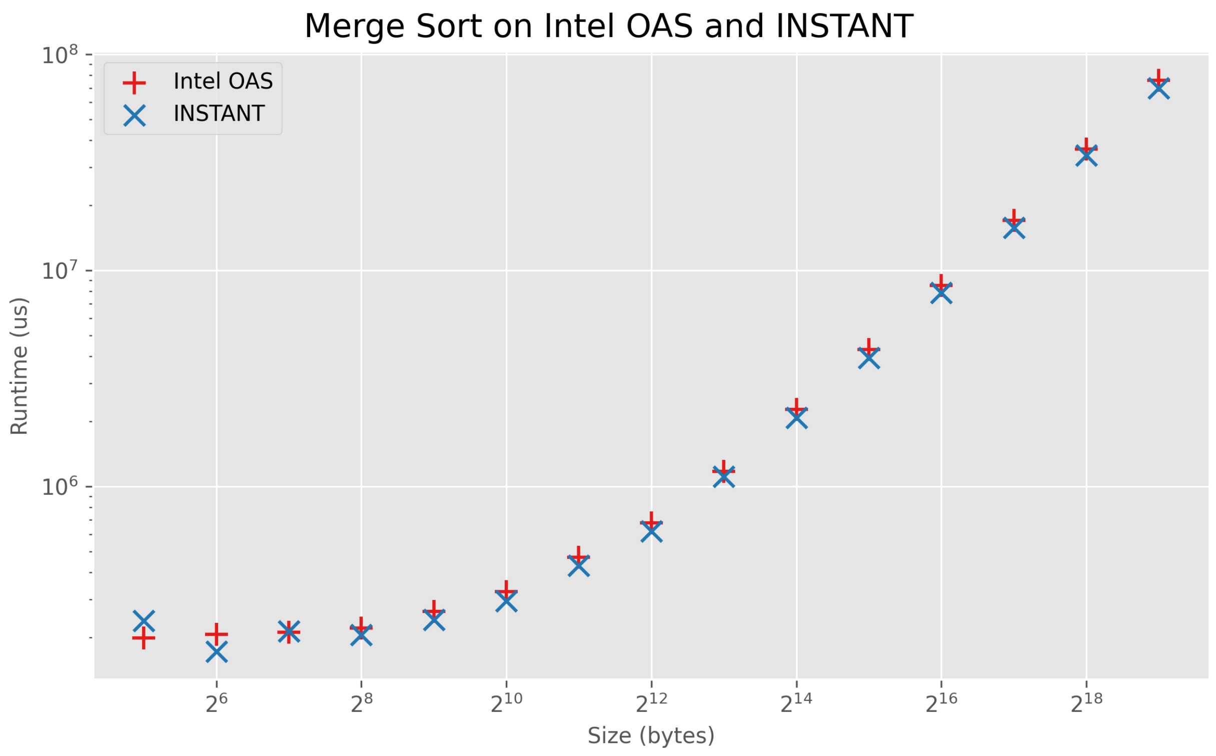

Figure 5.1: Runtime for the merge sort workload on the INSTANT system. All axes in log scale. The experiment was run with 2 disks and 1 GBPS as the network link speed.

suitable for setting up an NVMe-oF target. The host and each of the storage devices are their own instances. The experiments in this section aim to 1) measure the runtime of INSTANT compared to Intel OAS 2) find the conditions where peer to peer transfers are most beneficial 3) verify that INSTANT scales as the number of storage devices increases and 4) measure the overhead that INSTANT has over the standard Intel OAS.

5.2.1 Data Size

To study how the runtime scales with increasing size we performed a variety of experiments using different data sizes. The experiment was performed with 2 storage devices and the link speed between the hosts and disks was 1 GBPS. The results can be seen in Figure 5.1.

From the figure we can see that the runtime scales linearly with size, with the exception of the first few small data points.

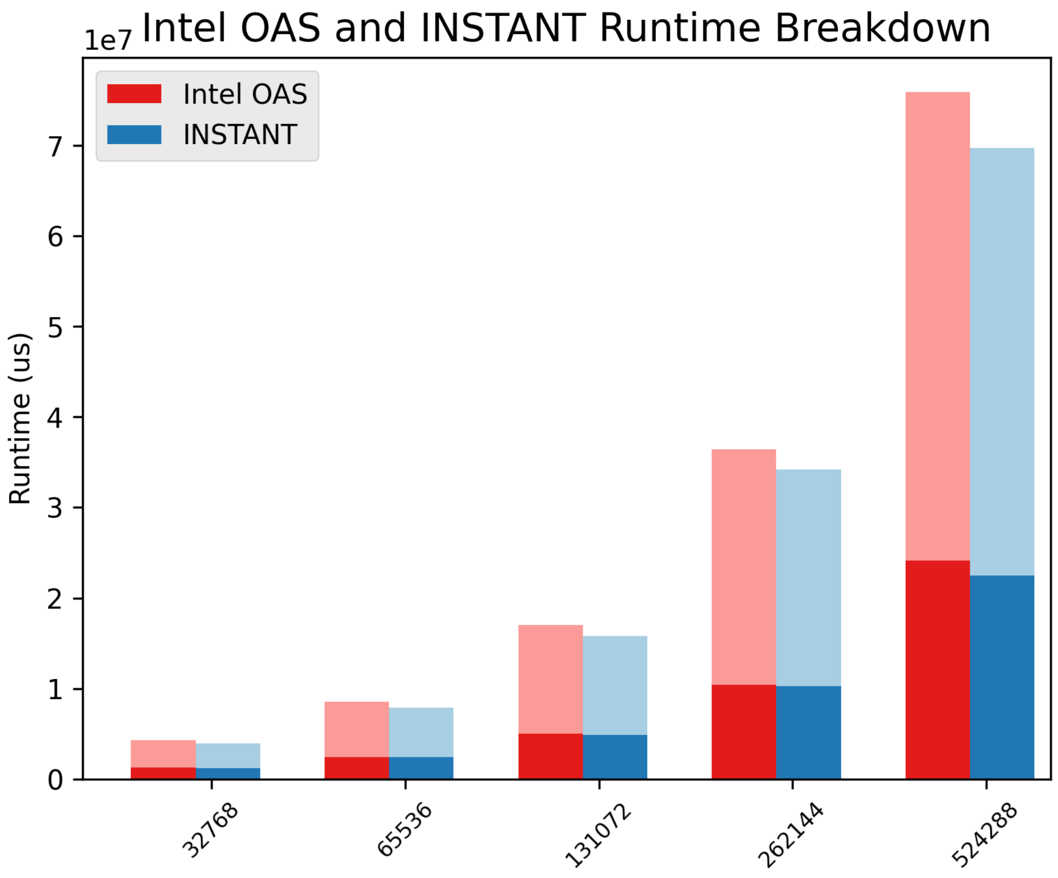

Figure 5.2: Runtime breakdown for the merge sort workload of Intel OAS. Y axis multiplied by 10^7. The experiment was run with 2 disks and 1 GBPS as the network link speed. Darker shade represents sort time and the lighter shade represents the merge time.

In that region we see relatively similar runtime which can be chalked down to the software stack overhead to initiate a computational storage command. We can see from Figure 5.1 that the INSTANT system's runtime increases in a similar fashion to Intel OAS with increased data sizes. Figure 5.2 focuses on larger data sizes where INSTANT has a meaningfully shorter runtime than Intel OAS. The breakdown between how much time is spent sorting and merging makes it clear that the gains are primarily made in the merge phase of the process. This is the expected as avoiding the trip to the host helps reduce the latency of the data transfer process.

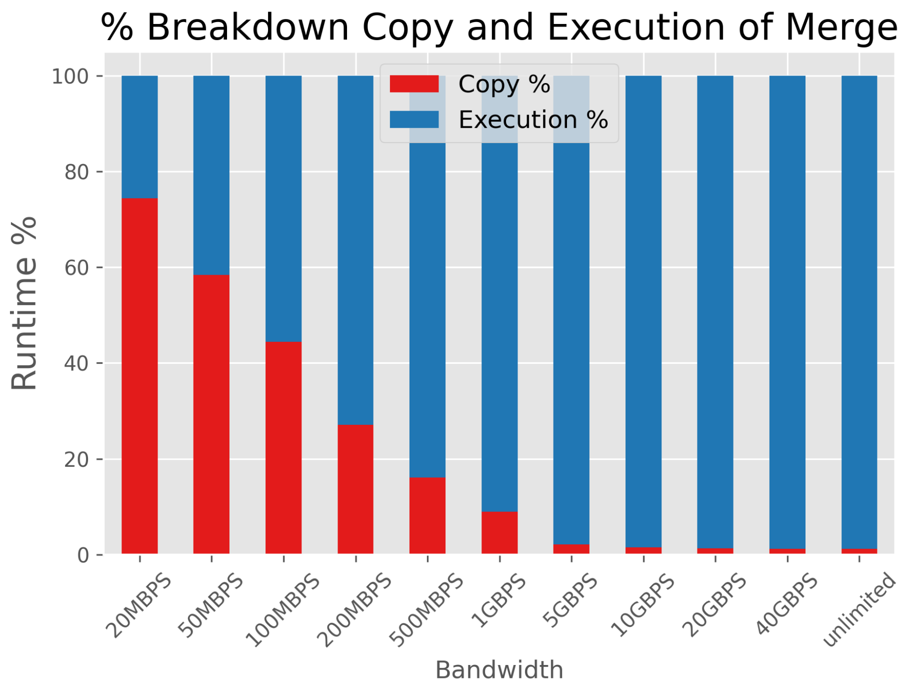

Figure 5.3: Runtime % Breakdown between Copy and Execution of Merge. The experiment was run with 2 disks and data size of 512 MB.

5.2.2 Network Link Bandwidth

As described in Chapter 4.2, the sort aspect of both INSTANT and Intel OAS is identical. As such, in this section we primarily focus on merge stages of the merge sort workload. This is because the merge stage is the part of the process where data is moved over the link itself. A sort stage is performed locally with data that is present on the disk itself. We experiment multiple data sizes using two storage devices under various different link speeds. Figure 5.3 shows that at lower bandwidths, the bottleneck in the overall merge runtime is the data transfer from one storage device to another and not the merge operation executed on the storage device and that as the link speed increases this botteneck shifts to the operation's execution. This is because the bandwidth of the merge operation is constant but the bandwidth of the data movement is affected by the bandwidth of the link. In Figure 5.5 we can also see that small sizes between 64KB and 4096KB consistently show a lower runtime on the INSTANT system. The overhead of initiating a separate copy adds latency in the Intel OAS system, and this is not required in the INSTANT system as the computational storage device can read data from another device directly. Finally, we also confirm that in a bandwidth constrained environment, disk to disk communication allows for a faster runtime as we reduce the number of times the data has to cross the link. We can also see from Figure 5.5 that the benefit of the disk to disk data transfer reduces as the link becomes less of a bottleneck at higher bandwidth speeds. At higher link speeds the runtime for the merge stage levels out. There is not much performance to be gained, if any, from INSTANT in situations where the data movement across the link is not a bottleneck.

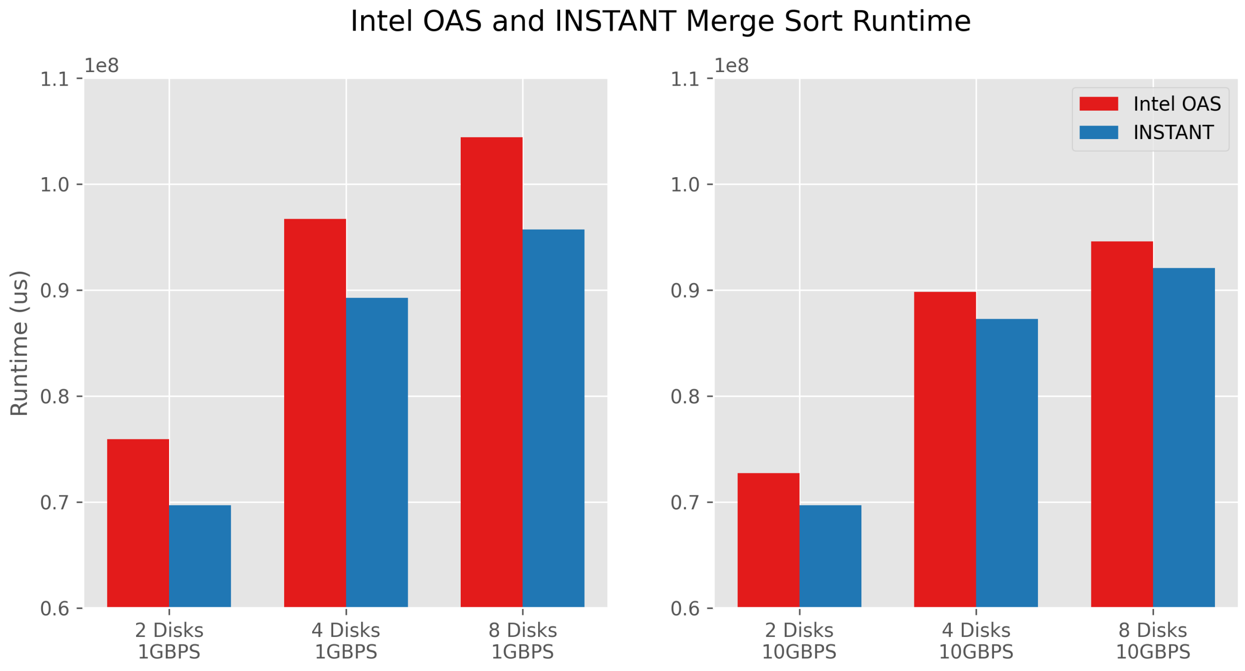

Figure 5.4: Runtime comparison with changing number of disks. Y axis is multiplied by 10^8. The experiment was run data size of 512 MB.

5.2.3 Number of Disks

One of the primary objectives of this experiment was to demonstrate that the system scales well as the number of devices increase. The experiment run in Figure 5.4 is a full 512MB merge sort at 1 GBPS and 10 GBPS as the link speed. In this figure, we can see that INSTANT consistently has a lower runtime than Intel OAS, even as the number of devices increases and that the trend for INSTANT is similar to that of Intel OAS. Considering the 1GBPS link speed, in the 2 disk system, INSTANT is 8.81% faster than Intel OAS, in the 4 disk system 8.60% faster than Intel OAS and in the 8 disk system 7.80% faster. Considering the 10GBPS link speed, in the 2 disk system, INSTANT is 4.20% faster than Intel OAS, in the 4 disk system 2.84% faster than Intel OAS and in the 8 disk system 2.65% faster. This trend where INSTANT's runtime gets closer to Intel OAS's runtime is due to the fact that there are more merges and smaller data size transfers as the number of disks increases, as such the data transfer over the link is a smaller percent of the overall merge runtime. From Figure 5.3 we can expect smaller gains as the bandwidth increases and this is exactly what we observe in this experiment as well.

5.2.4 Overhead

In this section we try to understand the overhead of INSTANT compared to the standard Intel OAS implementation. In Figure 5.2 we can see that merge operation is generally faster on the INSTANT system. To understand the overhead of communicating with COSTA instead of reading the local disk directly, we take a look at the sort operation. In Table 5.1 we can see a comparison of the sort stage runtime for INSTANT compared to Intel OAS. The sort stage for the smallest data sizes has most significant slowdown for INSTANT. We can attribute this to the specific implementation of COSTA. In INSTANT, COSTA is a separate process that receives a request from Harvest and performs a read and passes the read data back to Harvest. For small data sizes, the overhead of these interactions can be

Size	20 MBPS	50 MBPS	100 MBPS	200 MBPS	500 MBPS	1 GBPS	5 GBPS	10 GBPS	20 GBPS	40 GBPS	Unlimited
32KB	74.54	49.10	22.71	64.47	66.17	45.45	108.77	67.44	62.64	81.50	-14.24
64KB	-15.32	14.60	-8.29	-15.34	19.53	-13.03	-13.85	7.58	-7.05	5.56	28.24
128KB	30.58	-15.11	5.83	6.67	7.62	41.82	1.49	10.78	-6.60	1.46	9.01
256KB	11.40	20.18	17.46	5.72	-10.07	-12.27	31.88	-13.15	3.14	22.10	-11.38
512KB	-11.72	-0.94	-8.86	11.67	0.29	1.22	-11.46	22.92	-22.79	0.72	15.39
1MB	23.18	14.98	6.85	-8.23	-10.59	-3.16	12.09	29.34	33.25	-12.19	1.65
2MB	0.63	-12.63	19.61	15.18	0.86	-0.34	3.18	-6.98	-10.33	7.48	-8.90
4MB	-2.30	6.58	-7.31	1.80	0.51	-3.91	8.32	13.90	-0.62	14.64	5.61
8MB	-1.35	-3.32	5.53	3.04	-0.59	1.37	-6.82	-1.52	0.40	-5.18	0.48
16MB	2.38	2.90	-0.87	-0.46	1.87	0.60	4.33	-2.07	4.09	0.44	-4.06
32MB	5.11	1.43	0.99	0.83	-2.79	-4.11	3.76	0.81	5.88	-0.77	-0.79
64MB	-0.82	1.99	-0.26	-2.71	-1.15	1.04	-1.97	0.30	2.79	-4.99	0.43
128MB	-1.72	-0.94	-0.96	1.56	-1.55	-1.71	0.61	2.40	-3.71	-0.88	2.77
256MB	-3.00	-2.35	0.55	-2.03	-3.24	-1.62	2.24	3.06	-2.94	-1.09	1.04
512MB	-3.66	-0.08	-1.59	-0.01	0.43	-6.94	-5.94	-8.02	-5.27	-4.63	-0.03

Table 5.1: Comparing sort stage runtime overhead using percentage change in the runtime of INSTANT compared to Intel OAS. Data collected from 2 disks experiment.

more significant than for larger data transfers. However, these slowdowns are generally made up for during the merge stage as we can see that INSTANT generally performs better or as good as Intel OAS in most cases (Figure 5.5).

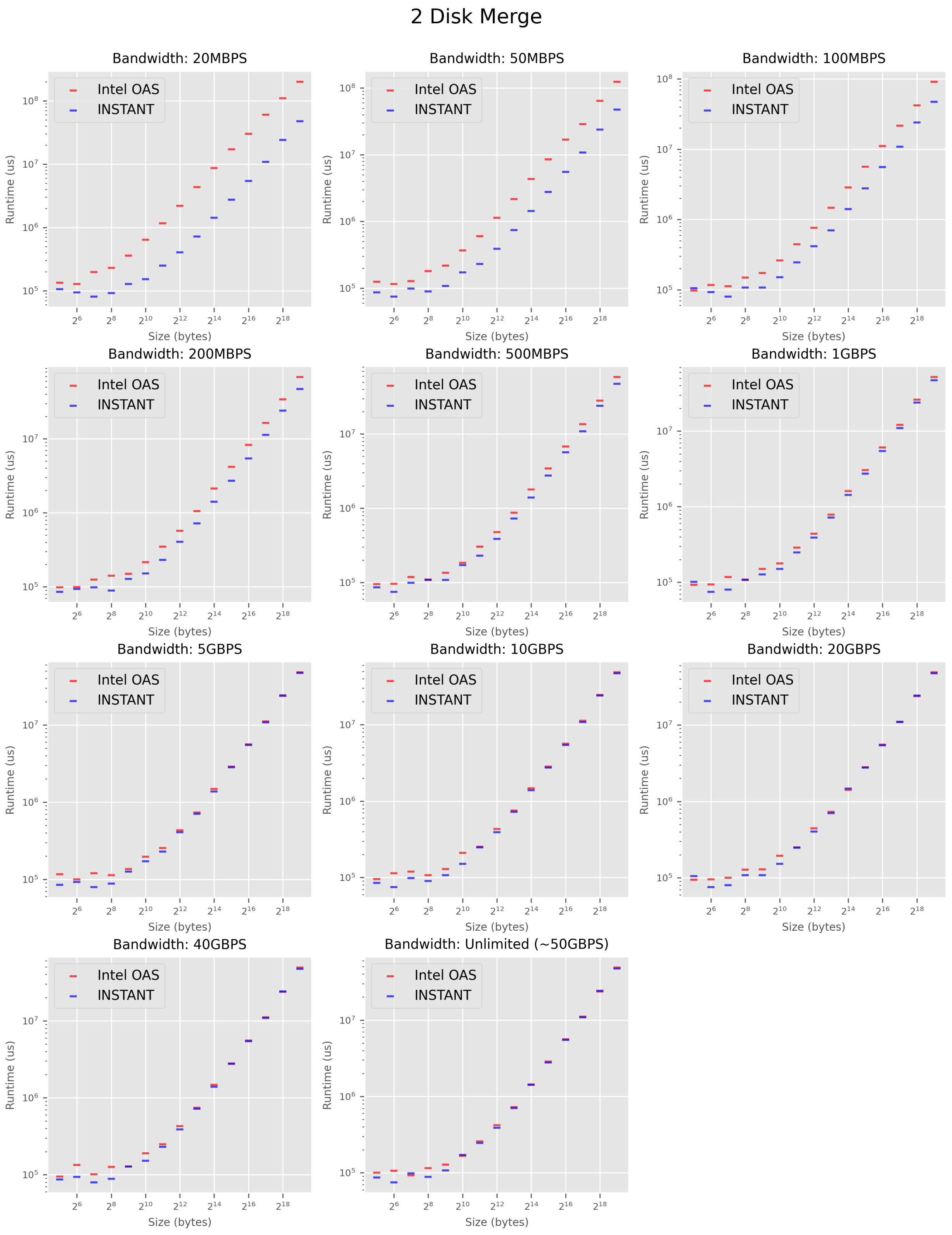

Figure 5.5: Runtime for Merge stage. All axes are log scale.

CHAPTER 6

Conclusion

6.1 Summary

From the experimentation, we can see that communication directly from one computational storage device to another can significantly improve the overall runtime especially in situations where bandwidth is the limiting factor. Even as the number of storage devices increases, we can see that communicating directly from device to device is still beneficial. We can see practical uses of this system in setups where there is a large number of storage devices all connected to one host - for example, in a JBOF (Just-a-Bunch-Of-Flash) server which has many SSDs connected to a single server. Such a setup would greatly benefit from processing that is available on each individual storage device as it will help not only parallelize computation, but with device to device communication, lower the amount of data moved around during the processing stage.

6.2 Optimizations

The following section will describe potential optimizations that can improve the overall performance of INSTANT.

6.2.1 Optimization 1: Improving Intel OAS Harvest

We can see that one of the more peculiar aspects from the evaluation section is Figure 5.3. In this figure, we observe that the merge kernel execution takes a very long time to be performed in comparison to the data transfer. We know that merging is not a computationally intensive task and that data movement should be the majority of the process. Intel OAS Harvest has a few inefficiencies in the way a merge is performed on the remote server and these can be improved vastly. The current implementation of merge on Intel OAS Harvest writes input data to a temporary filesystem before reading it and then finally writing the result to the NVMe SSD using the disk dump utility. Performing the operation in memory and writing to the SSD directly from Harvest would vastly improve the performance of the merge kernel.

6.2.2 Optimization 2: NVMe Protocol Extension Supporting Disk to Disk Communication

This optimization applies to SCANCOMS as a whole and not just INSTANT. Having the core mechanism to enable disk to disk communication within the NVMe protocol itself would allow for a standardized implementation that enables disk to disk communication. The NVMe protocol already has several concepts that can be extended to integrate such a mechanism. The idea of a namespace in NVMe encapsulates multiple devices under a single entity. This notion can be extended to expose more APIs to allow applications to have visibility into source and destination devices within a namespace.

6.2.3 Optimization 3: TCP Tuning for Performance

During the experiments, the amount of data transferred between devices was at minimum a several kilobyte chunk. We can tune the TCP parameters to have larger buffer sizes and support larger windows to facilitate the larger data transfers with less interference from the operating system's networking stack. We can also switch to a lower latency networking stack

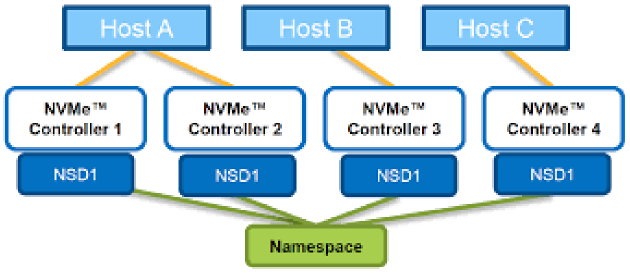

Figure 6.1: NVMe Namespace Sharing [Wor19a].

such as a DPDK based network driver which can be integrated with the userspace software without any kernel interference.

6.2.4 Optimization 4: Implement High Performance FS

While FUSE is extremely convenient, it is not a filesystem that is very high performance. There are many limitations on the buffer sizes that it can support and many redundant allocations/copies that occur during any given file IO operation involving FUSE. A higher performance filesystem would eliminate such allocations/copies and allow for larger buffers to be transferred, thereby reducing the overall latency of the filesystem. While the exact impact of FUSE is hard to measure, we did have to make modifications to FUSE to handle requests for sizes larger than 1MB at a time.

6.3 Future Work

While some of these optimizations can be achieved with a little bit of effort, we find that the optimizations don't help us develop a better understanding of computational storage across multiple devices. In order to do that, we propose some future work ideas that will help use push the envelope further and hopefully provide new and interesting insight in this domain.

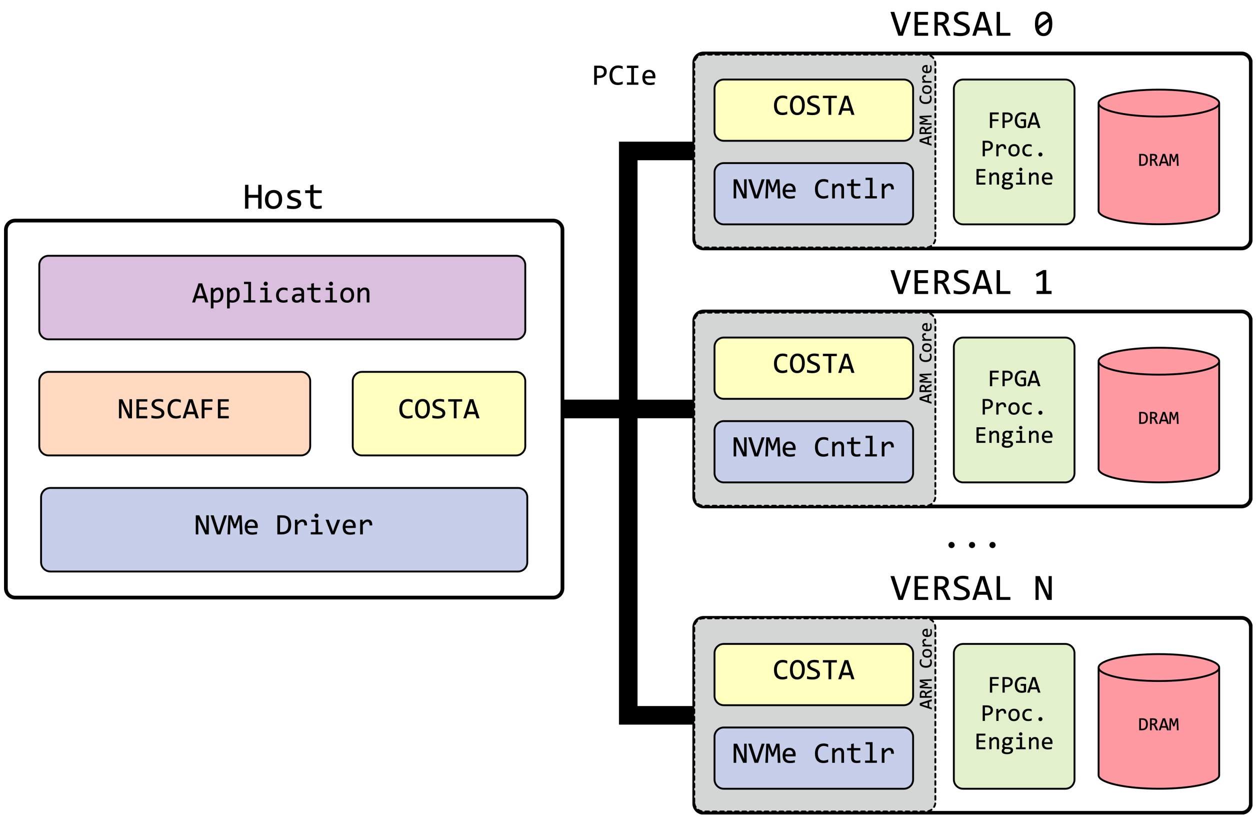

Figure 6.2: SCANCOMS implemented using an FPGA as the Storage Device Emulator.

6.3.1 Future Work 1: SCANCOMS on a Cluster of Versal Boards

Taking inspiration from INSIDER [RHC19b], implementing SCANCOMS on an Versal board would be extremely beneficial to studying the impact of having several computational storage devices work together. The Versal board [Xil20] is an adaptive compute acceleration platform that provides an FPGA, ARM CPUs and specialized hardware accelerators all in one chip connected by a high bandwidth network on chip. SCANCOMS on such a system, visualized in Figure 6.2, would consist of the Versal boards emulating storage device with DRAM, running the NVMe Controller and COSTA softwares on the ARM CPU and the in storage processors implemented on the FPGA. This type of platform would allow for experimentation on speeds much closer to real PCIe speeds and also explore the impact of hardware acceleration on near storage workloads. A cluster of FPGAs could be directly connected to each other providing a very high bandwidth, low latency platform to study how several of these devices can be used effectively using SCANCOMS.

To impelement SCANCOMS on such a system we would have to update NESCAFE to interface with the INSIDER [RHC19b] system instead of the Intel OAS system - which also means that the NVMe-oF protocol will have to be dropped in favor for the INSIDER protocol. The NESCAFE implementation on INSTANT is modular, however, a significant portion of the code base will have to be modified to use the INSIDER API instead. The bigger challenge, however, lies in porting COSTA to the Versal board. The core of COSTA's business logic is platform independent however the communication module will have to be changed from a network based interface to something that will allow communication to and from the ARM processor on the Versal board. This will likely be have to performed over PCIe. There are a variety of implementation options here: 1) A shared memory approach where a region in the address space maps to different Versal devices. 2) Using the tun driver to emulate a network over PCIe. The tun driver can allow a shared pool of memory to be used by the networking stack and expose a network interface to applications. This approach might require fewer changes to port the INSTANT implementation of COSTA to the Versal board at the expense of adding a higher overhead for transferring data, albeit incurring the overhead of the networking stack.

6.3.2 Future Work 2: Additional Workloads

Another aspect of computational storage that needs exploration is to understand what types of workloads truly reap the benefits of computational storage. There is a comprehensive exploration of this done by [RHC19a] but we can extend this work by looking at workloads that greatly benefit from having multiple computational storage devices. We can study workloads such as graph matching, machine learning or database applications such as log structured merge (LSM) tree compaction. LSM tree compaction is a very IO heavy operation and if offloaded to the disk can help speed up popular databases such as RocksDB and

LevelDB. Graph matching and machine learning are operations that require a lot of input data but also have a lot of intermediate state that is created and one device's computation may be dependent on another device. This leads us to our last future work idea.

6.3.3 Future Work 3: State Transfer vs Data Transfer

In many cases, transferring the data from one device to another is the most beneficial solution, however, when the amount of computational state created is much smaller than the size of the data being operated on, transferring the current state of the computation may result in greater savings of link bandwidth and provide an overall acceleration. We can explore the types of workloads that can most benefit from such a transfer and also try to understand the feasibility of implementing a system that supports this in a user friendly manner.

REFERENCES

[AKM19] Ian F. Adams, John Keys, and Michael P. Mesnier. "Respecting the Block Interface - Computational Storage Using Virtual Objects." In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, p. 10, USA, 2019. USENIX Association.

[AUS98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation." In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, p. 81–91, New York, NY, USA, 1998. Association for Computing Machinery.

[CPO13] 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*, ICS '13, p. 91–102, New York, NY, USA, 2013. Association for Computing Machinery.

[CS13] Adrian M. Caulfield and Steven Swanson. "QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks." In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, p. 464–474, New York, NY, USA, 2013. Association for Computing Machinery.

[CSM14] M. Chadalpaka, J. Staran, K. Meth, and D. Black. "Internet Small Computer System Interface (iSCSI) Protocol." RFC 7143, RFC Editor, April 2014.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Commun. ACM*, **51**(1):107–113, January 2008.

[DG17] Jeffrey Dean and Sanjay Ghemawat. "LevelDB.", 2017.

[GYB16] 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)*, pp. 153–165, 2016.

[J 04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. "Internet Small Computer Systems Interface (iSCSI)." RFC 3720, RFC Editor, April 2004.

[JLL15] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. "BlueDBM: An Appliance for Big Data Analytics." In

Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, p. 1–13, New York, NY, USA, 2015. Association for Computing Machinery.

[JWZ18] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. "GraFboost: Using Accelerated Flash Storage for External Graph Analytics." In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, p. 411–424. IEEE Press, 2018.

[JXA17] Sang-Woo Jun, Shuotao Xu, and Arvind. "Terabyte Sort on FPGA-Accelerated Flash Storage." In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 17–24, 2017.

[KMI17] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. "Summarizer: Trading Communication with Computing near Storage." In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, p. 219–231, New York, NY, USA, 2017. Association for Computing Machinery.

[KPH98] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. "A Case for Intelligent Disks (IDISKs)." *SIGMOD Rec.*, **27**(3):42–52, September 1998.

[OLJ14] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. "SDF: Software-Defined Flash for Web-Scale Internet Storage Systems." In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, p. 471–484, New York, NY, USA, 2014. Association for Computing Machinery.

[RHC19a] Zhenyuan Ruan, Tong He, and Jason Cong. "Analyzing and Modeling In-Storage Computing Workloads On EISC — An FPGA-Based System-Level Emulation Platform." In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019.

[RHC19b] Zhenyuan Ruan, Tong He, and Jason Cong. "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive." In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 379–394, Renton, WA, July 2019. USENIX Association.

[RHL18] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. "ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA." In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*, pp. 9–16. IEEE Computer Society, 2018.

[SGB14] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. "Willow: A User-Programmable SSD." In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, p. 67–80, USA, 2014. USENIX Association.

[SPW20] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. "Accessible Near-Storage Computing with FPGAs." In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[TZZ16] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. "Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing." *SIGARCH Comput. Archit. News*, **44**(3):53–65, June 2016.

[Wor19a] NVM Express Workgroup. "NVM Express 1.4 Spec.", Jun 2019.

[Wor19b] NVM Express Workgroup. "NVM Express over Fabrics 1.1 Spec.", Oct 2019.

[WSJ14] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. "An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD." In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[XBH20] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. "AQUOMAN: An Analytic-Query Offloading Machine." In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 386–399, 2020.

[Xil20] Xilinx. "Versal, the First Adaptive Compute Acceleration Platform (ACAP).", Sep 2020.

[XLJ16] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. "Bluecache: A Scalable Distributed Flash-Based Key-Value Store." *Proc. VLDB Endow.*, **10**(4):301–312, November 2016.