Improving Data Management and Data Movement Efficiency in Hybrid Storage Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Xiongzi Ge

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

David H.C. Du

July, 2017

Acknowledgements

There are numerous people I would like to express my gratitude for their impact on me and the contributions to my research over the years. First of all, I would like to thank my advisor Professor David H.C. Du for supervising me over the past nine years, both on site and remotely, and thus I grow personally and professionally. I also would like to thank the other committee members, Prof. Tian He, Prof. Rui Kuang and Prof. Soheil Mohajer for their flexibility and insightful suggestions.

I would like to thank Dennis Hahn and Pradeep Ganesan from NetApp for their mentorship. Special thanks go to Yi Liu and Liang Zhang from Huawei, Chengtao Lu, Zhichao Cao and Xuchao Xie for the collaboration and friendship. I would like to thank the sponsorship and the members I met in the Center for Research in Intelligent Storage (CRIS) for their help: Xiang Cao, Jim Diehl, Ziqi Fan, Alireza Haghdoost, Weiping He, Bingzhe Li, Manas Minglani, Yaobin Qin, Fenggang Wu, Hao Wen, Jinfeng Yang, Baoquan Zhang and Meng Zou. In particular, I would like to thank the support from National Science Foundation (NSF awards 1305237, 1421913, 1439622 and 1525617).

I am grateful to the Advanced Technology Group in NetApp for their cutting-edge research and the brilliant researchers. I would like to give special thanks to my manager Albert Andux, Andrew Klosterman, John Strunk, Chris Dragga and Ardalan Kangarlou for their help when I had to cope with job, thesis and life in a new place.

Finally, I especially would like to thank my family. Without them, I could not reach this far.

Dedication

To my family.

Abstract

In the big data era, large volumes of data being continuously generated drive the emergence of high performance large capacity storage systems. To reduce the total cost of ownership, storage systems are built in a more composite way with many different types of emerging storage technologies/devices including Storage Class Memory (SCM), Solid State Drives (SSD), Shingle Magnetic Recording (SMR), Hard Disk Drives (HDD), and even across off-premise cloud storage.

To make better utilization of each type of storage, industries have provided multi-tier storage through dynamically placing hot data in the faster tiers and cold data in the slower tiers. Data movement happens between devices on one single device and as well as between devices connected via various networks. Toward improving data management and data movement efficiency in such hybrid storage systems, this work makes the following contributions:

To bridge the giant semantic gap between applications and modern storage systems, passing a piece of tiny and useful information (I/O access hints) from upper layers to the block storage layer may greatly improve application performance or ease data management in heterogeneous storage systems. We present and develop a generic and flexible framework, called HintStor, to execute and evaluate various I/O access hints on heterogeneous storage systems with minor modifications to the kernel and applications. The design of HintStor contains a new application/user level interface, a file system plugin and a block storage data manager. With HintStor, storage systems composed of various storage devices can perform pre-devised data placement, space reallocation and data migration polices assisted by the added access hints.

Each storage device/technology has its own unique price-performance tradeoffs and idiosyncrasies with respect to workload characteristics they prefer to support. To explore the internal access patterns and thus efficiently place data on storage systems with fully connected (i.e., data can move from one device to any other device instead of moving tier by tier) differential pools (each pool consists of storage devices of a particular type), we propose a chunk-level storage-aware workload analyzer framework, simplified

as ChewAnalyzer. With ChewAnalzyer, the storage manager can adequately distribute and move the data chunks across different storage pools.

To reduce the duplicate content transferred between local storage devices and devices in remote data centers, an inline Network Redundancy Elimination (NRE) process with Content-Defined Chunking (CDC) policy can obtain a higher Redundancy Elimination (RE) ratio but may suffer from a considerably higher computational requirement than fixed-size chunking. We build an inline NRE appliance which incorporates an improved FPGA based scheme to speed up CDC processing. To efficiently utilize the hardware resources, the whole NRE process is handled by a Virtualized NRE (VNRE) controller. The uniqueness of this VNRE that we developed lies in its ability to exploit the redundancy patterns of different TCP flows and customize the chunking process to achieve a higher RE ratio.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivations

Traditional spinning hard disk drives (HDDs) dominate the storage world for more than half a century. Over the past decade, several new storage technologies emerge and become mature. For examples, Seagate recently announced the Perpendicular Magnetic Recording (PMR) based enterprise 12 TB drives [1]. The Shingle Magnetic Recording (SMR) [2] technology increases the areal density by overlapping adjacent write tracks. Flash-based Solid State Drives (SSDs)[3] make storage incredibly faster. Today, there are even more high performance storage devices/ technologies like phase change-memory (PCM), spin-torque transfer RAM (STT-RAM) and memristors [4] that not only offer non-volatile memory [5] but also promise high performance non-volatile storage [6]] (termed Storage Class Memory (SCM)). In addition, cloud storage (e.g. Amazon S3 and Azure) emerges as a mainstream for reducing cost and increasing data reliability.

Each of the above devices/technologies has its own unique performance characteristics. Table 1.1 shows the summary of a set of the off-the-shelf storage technologies. SSDs and SCM fundamentally differ from the traditional HDDs. They can read/write data randomly without moving the mechanical arm and heads. In addition, flash based SSDs have to erase a data block before serving new incoming write requests to the block. This makes SSDs have asymmetric read/write speeds. SCM like Phase Change Memory (PCM) and STT-RAM is referred as a high-speed, non-volatile storage device. Although SCM has an impressive performance (hundreds of thousands IO per second),

Table 1.1: Summarization of diverse storage devices/techniques.

Storage	**Throughput**	**IOPS**	**Pros vs. Cons**
Intel 3D XPoint Technology	1000x faster than NAND Flash	1000x faster than NAND Flash	The highest performance vs. high price, low density
Samsung 950 PRO NVMe SSD (PCI-e) 256GB	Read:2200MBps Write:900MBps	Read: 270 000 Write:85 000	High performance but high price vs. limited write times
Samsung 850 PRO 2.5S̈SD SATA III 512GB	Read:550MBps Write:220MBps	Read:10 000 Write:90 000	Better performance than HDD vs. limited write times
Seagate 600GB 15k SAS HDD (ST600MP0005)	160-233MBps	∼500 (latency: 2ms)	High density vs. low performance
Seagate SMR Archive HDD 8TB	150MB/s	∼200 (latency:5.5ms)	Cheap, high density vs. low performance
IBM TS1150 tape drive 4TB	360MBps	Very low	Cheap, high density, long-term preservation vs. low performance

its cost is still very expensive, even exceeding that of a many-core CPU [6]. For cloud storage, the latency over the internet is 100-1000 times higher than that of accessing data from local servers [7]. For instances, both Amazon S3 and Azure have high GET latency (average 200ms or more per operation) [8]. This results from the long distance and unpredictable and disconcertingly inconsistent connection. More seriously, cloud latency may cause serious data corruption, data unreachable transiently and more terribly even data vanishings all of a sudden [7].

To reduce the total cost of ownership, storage systems are built in a more composite way incorporating the above emerging storage technologies/devices, including Storage Class Memory (SCM), Solid State Drives (SSD), Shingle Magnetic Recording (SMR) and even across off-premise cloud storage. This makes enterprise storage hierarchies more interesting and diverse. To make better utilization of each type of storage, industries have provided multi-tier storage through dynamically placing hot data in the faster

tiers and cold data in the slower tiers. Data movement happens between devices on one single device and as well as between devices connected via various networks. Managing data and achieving efficient data movement become challenging in such hybrid storage systems.

Firstly, the current storage stack is designed to mainly accommodate the properties of HDDs. Multiple independent layers are developed and exist in the storage stack in a hierarchical manner. For examples, the existing I/O stack provides a series of homogenous logical block addresses (LBAs) to the upper-level layers like filesystems, databases, or applications. A filesystem is always built on a virtualized logical device. Such a logical device may consist of several different types of storage devices (e.g. SSDs and HDDs). The block layer that managing the storage space lacks necessary information about file data, such as file metadata and data blocks belonging to the file. This makes block layer storage manager hard to appropriately allocate resources and place data blocks across different storage devices. The gap between applications and block storage is even larger for applications have no clue of what types of storage devices they are accessing. To narrow the gap between storage system and its upper layers, one of the promising approaches is using I/O access hints [9, 10]. When an application is reading a file, it may have no clue of the physical location of the file. This file may be scattered over several devices even with a portion of the blocks in a low performance device. If the upper layers can send hints to the storage system, the storage controller may proactively load the associated blocks of the requested files from a low-performance device to a high-performance tier.

The previous work related to I/O access hints mainly focuses on the two-layer storage architecture (memory and storage). A classifier was proposed in [10] which allows the storage controller to employ different I/O policies with each storage I/O command. For example, an SSD device can prioritize metadata and small files for caching in a file system. Sonam Mandal [9] studied block-layer data dedupliation by using hints from upper layers to guide the dmdedup engine to bypass certain types of data (e.g. metadata) and prefetch the associated index data of the data chunks. However, these investigations do not study storage layer data management like data migration across different devices. In addition, one of the major challenges of research in storage access hints is lacking a common platform for evaluation. In order to send hints to the

storage layer, different applications require to have an efficient way of communicating with storage layer (e.g. sysfs, SCSI commands). Developing hints and evaluating their effectiveness may need to change the applications, file systems and the block storage management software. This results in tedious work before a particular access hint can be evaluated for its effectiveness.

Secondly, in the past, storage tiering has been viewed as a method of getting both performance and affordability, however, moving data tier by tier may be not efficient in such hybrid storage systems. The legacy two-layer architecture, such as hybrid cache [11] combining flash-based SSD [12] and main memory, hybrid SSD and HDD storage [13], has been extensively studied. Data tiering is also used in local and cloud storage, such as FabricPool which is built on NetApp's ONTAP system combining with the remote on-premise cloud storage [14].The expected process of data placement is that fast tiers serve a group of intensive workloads for better performance while slow tiers are persistently storing the rest of data blocks to provide lower cost [15]. Data migration happens tier by bier when data access frequency changes. However, moving data tier by tier may not be efficient and even worse it may lead to unnecessary data movements.

Accurate chunk level workload characterization can help the system understand what resources are adequate for the associated requests. However, conventional storage workload analysis methods oriented to tier by tier cases are not applicable on multiple differentiated storage pools. The classical hot and cold data classification methodology is employed in the tier-by-tier case [16]. Each data chunk is characterized as certain pattern, typically, high or low IOPS in a period, and followed by a greedy migration policy to be moved between different storage pools [17]. However, employing different dimensions and granularities may generate entirely different access patterns. We investigate a set of enterprise block I/O workloads using different chunk sizes and different taxonomy rules. Typically, smaller chunk size incurs more metadata management while larger chunk size reduces the flexibility of data management. Different taxonomy rules may partition data into totally different categories. Moreover, workload profiling is highly related to the device technology. For example, it is valuable to pay more attention to write requests when both SSDs and SCM are available. Although both offer better random I/Os per second (IOPS) than HDDs, the not-in-place-update and lifetime issue still drive researchers and developers to reduce and even eliminate random write I/Os

on flash based SSDs [18, 19]. Furthermore, applications like big data processing (e.g. Hadoop) have their own characters (e.g. streaming or batch processing) [17].

Thirdly, as data is moving between local severs and remote cloud centers, a huge amount of duplicate information transferring causes network congestion. Network Redundancy Elimination (NRE) aims to improve network performance by identifying and removing repeated transmission of duplicate content from remote servers [20–22]. For an inline NRE process, the content of network flows is first segmented into a set of data chunks. Then these data chunks will be identified to be redundant (i.e., has been recently transmitted and buffered) or not. Generally, a higher Redundancy Elimination (RE) ratio can save more bandwidth [23]. Chunking policies based on either fixed or variable sizes determine the RE ratio [20]. Compared with a fixed-size chunking policy, a variable-size chunking policy can more efficiently identify repetitive chunks.

Content Defined Chunking (CDC) [24], a variable chunking policy, has been widely used by many NRE solutions [21, 25, 26]. However, some components of CDC consume significant CPU time (e.g., the Rabin hash process). For instance, considering a standard software-based NRE MB (Intel E5645 CPU, 2.4 GHz, 6 cores, exclusive mode), the CDC chunking throughput is about 267 Mbps for each core and totals around 1.6 *Gbps* [27]. This overhead will affect the server performance and eventually decrease the overall NRE throughput. For a CDC scheme, there is also a tradeoff between RE ratio and the expected data chunk size. The smaller the expected chunk size, the higher the RE ratio. However, the smaller expected chunk size will require higher computational cost.

1.2 Contributions

To improve data management and data movement efficiency in hybrid storage systems, this thesis mainly makes the following three contributiions:

- A generic and flexible framework, called HintStor, is proposed and developed in the current Linux system, to execute and evaluate various I/O access hints on heterogeneous storage systems with minor modifications to the kernel and applications. The design of HintStor contains a new application/user level interface,

a file system plugin and a block storage data manager. The block storage data manger implements four atomic access hints operations in Linux block level, which can perform storage layer data management like data replication and data migration. The file system plugin defines a file level data classification library for common file systems. A new application/user level interface allows users to define and configure new access hints. With HintStor, storage systems composed of various storage devices can perform pre-devised data placement, space reallocation and data migration polices assisted by the added access hints, such as file system data classification, stream ID and cloud prefetch.

- To explore the internal access patterns and thus efficiently place data on storage systems with fully connected (i.e., data can move from one device to any other device instead of moving tier by tier) differential pools (each pool consists of storage devices of a particular type), we propose a chunk-level storage-aware workload analyzer framework, simplified as ChewAnalyzer. Access patterns are characterized as a collective I/O accesses in a chunk composed of a set of consecutive data blocks. The taxonomy rules are defined in a flexible manner to assist detecting chunk access patterns. In particular, ChewAnalyzer employs a Hierarchical Classifier to exploit the chunk patterns step by step. With ChewAnalzyer, the storage manager can adequately distribute and move the data chunks across different storage pools.

- To reduce the duplicate content transferred between local storage devices and devices in remote data centers, we build an inline NRE appliance which incorporates an improved FPGA based scheme to speed up CDC processing. To efficiently utilize the hardware resources, the whole NRE process is handled by a Virtualized NRE (VNRE) controller. The uniqueness of this VNRE that we developed lies in its ability to exploit the redundancy patterns of different TCP flows and customize the chunking process to achieve a higher RE ratio. Through the differentiation of chunking policies for each flow, the overall throughput of the VNRE appliance is improved.

1.3 Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents HintStor, a generic and flexible framework to study I/O access hints in heterogeneous storage systems.
- Chapter 3 describes ChewAnalyzer, which targets workload-aware data management across differentiated storage pools
- Chapter 4 shows VNRE, to achieve flexible and efficient acceleration for NRE.
- Chapter 5 concludes the thesis.

Chapter 2

A Flexible Framework to Study I/O Access Hints in Heterogeneous Storage Systems

To bridge the giant semantic gap between applications and modern storage systems, passing a piece of tiny and useful information (I/O access hints) from upper layers to the block storage layer may greatly improve application performance or ease data management in storage systems. This is especially true for heterogeneous storage systems. Since ingesting external access hints will likely involve laborious modifications of legacy I/O stacks, thus making it is very hard to evaluate the effect of access hints. In this chapter, we present and develop a generic and flexible framework, called HintStor, to quickly play with a set of access hints and evaluate their impacts on heterogeneous storage systems.

2.1 Introduction

Conventional hard disk drives (HDDs) dominate the storage world for more than half a century. In the past decade, several new storage technologies emerged and became mature. Perpendicular Magnetic Recording (PMR) aligns the poles of each magnetic

element vertically and thus allows manufacturers to reduce the size of each bit and increase the capacity of a single HDD. For example, Seagate recently announced the PMR based enterprise 12 TB drives [1]. Shingle Magnetic Recording (SMR) [2] technology increases the areal density by overlapping adjacent write tracks. For example, Western Digital plans to announce the 14TB SMR based HDDs [28]. Both PMR and SMR are the variants of HDDs. Besides, some high-performance storage devices, like flash-based Solid State Drives (SSDs)[3] and Storage-level Class Memory (SCM)[6] make storage faster. SSDs and SCM fundamentally differ from the traditional HDDs. They can read/write data randomly without moving the mechanical arm and heads. In addition, flash based SSDs have to erase a data block before serving new incoming write requests to the block. This makes SSDs have asymmetric read/write speeds. SCM like Phase Change Memory (PCM) and STT-RAM is referred as a high-speed, non-volatile storage device. Although SCM has an impressive performance (hundreds of thousands IO per second), its cost is still very expensive, even exceeding that of a many-core CPU [6]. Taking into consideration of performance, capacity and cost, storage systems are built in a more composite way incorporating the emerging storage technologies/devices, including SCM, SSD, SMR and even across off-premise cloud storage. We consider this type of storage systems heterogeneous.

The current storage stack is designed to mainly accommodate the properties of HDDs. Multiple independent layers are developed and exist in the storage stack in a hierarchical manner. For examples, the existing I/O stack provides a series of homogenous logical block addresses (LBAs) to the upper-level layers like filesystems, databases, or applications. A filesystem is always built on a virtualized logical device. Such a logical device may consist of several different types of storage devices (e.g. SSDs and HDDs). The block layer that managing the storage space lacks necessary information about file data, such as file metadata and data blocks belonging to the file. This makes block layer storage manager hard to appropriately allocate resources and place data blocks across different storage devices. The gap between applications and block storage is even larger for applications have no clue of what types of storage devices they are accessing. For example, a maintenance task, like backup or layout optimization, which may be designed to improve data availability or system performance [29]. Such maintenance I/O requests may be mixed with the foreground I/O requests. Storage systems should

prioritize the latter ones if they can. In addition, those backup data blocks are good candidates to reside on low speed storage devices. However, when I/O requests arrive at the block layer, the storage controller usually is not able to fully recognize the data properties and differentiate them from the foreground requests. To balance the storage capacity and performance, in a tiered-storage the storage manager tries to move cold data from a fast-tier to a slow-tier after the data blocks become less frequently accessed. The decision is mainly based on the statistics of data access frequency. If the storage manager moves cold data too conservatively, the I/O latency may suffer when there is not sufficient space for hot data to be migrated to the fast-tier. On the other hand, an aggressive data migration may lead to unnecessary data traffics between layers and eventually degrade the overall performance. Note that we are using fast-tier and slow-tier to simplify our discussion. A heterogeneous storage system may consist of several types of storage devices with different performance and access properties.

To narrow the gap between storage system and its upper layers, one of the promising approaches is using I/O access hints [9, 10]. When an application is reading a file, it may have no clue of the physical location of the file. This file may be scattered over several devices even with a portion of the blocks in a low performance device. If the upper layers can send hints to the storage system, the storage controller may proactively load the associated blocks of the requested files from a low-performance device to a high-performance tier. A classifier was proposed in [10] which allows the storage controller to employ different I/O policies with each storage I/O command. For example, an SSD device can prioritize metadata and small files for caching in a file system. Sonam Mandal [9] studied block-layer data deduplication by using hints from upper layers to guide the dmdedup engine to bypass certain types of data (e.g. metadata) and prefetch the associated index data of the data chunks. However, these investigations do not study storage layer data management like data migration across different devices. In addition, one of the major challenges of research in storage access hints is lacking a common platform for evaluation. In order to send hints to the storage layer, different applications require to have an efficient way of communicating with storage layer (e.g. sysfs, SCSI commands). Developing hints and evaluating their effectiveness may need to change the applications, file systems and the block storage management software. This results in tedious work before a particular access hint can be evaluated for its

effectiveness.

In this chapter, we design a generic and flexible framework, called HintStor, to study access hints in heterogeneous storage systems. The purpose of HintStor is not trying to design and deliver a dozen of access hints mechanisms. The major goal is devising a framework for access hints evaluation. The major goal is devising a framework for access hints evaluation. To perform access hints in storage systems, we design and implement a new application/user level interface, a file system plugin and a block storage data manager in HintStor. The block storage data manger implements four atomic access hints operations in Linux block level, which can perform storage layer data management like data replication and data migration. The file system plugin defines a file level data classification library for common file systems. A new application/user level interface allows users to define and configure new access hints. With HintStor, we can quickly play with various access hints and evaluate their efficiency and effectiveness on storage level data management.

Generally, HintStor divides the access hints into two categories, static hints and dynamic hints. The model is not only relevant to the T-10 SCSI hint proposal from Frederick Knight et al.[30], but also extend to user defined APIs not restricted to SCSI commands. In the static model, the information of I/O access hints contains the file data information (e.g. metadata vs. data blocks, small size file vs. large size file) like the legacy hints model. HintStor calculates the ratio of different types of storage devices and makes data placement decision at the block layer with the help of hints. With the dynamic model, HintStor monitors the real-time data access and usage, and aided by applications to provide dynamic hints to the storages system. In addition, application level interface allows applications to define storage requirements for the data (e.g. low latency, backup data, archival data, etc.). HintStor triggers data migration aided by application hints and the block-level statistics (e.g. heatmap).

We implement a prototype of HintStor in Linux 3.13.0. All implemented interfaces only require few modifications to file systems and applications. Various access hints can be implemented and evaluated quickly using HintStor. To show the flexibility of our framework, we evaluate different configurations with various storage (e.g. SSD, HDD, SCM and cloud based storage). Specifically, we play and evaluate the following access hints using HintStor:

- File system internal data classification: We use internal file system information to classify the data via different dimensions like metadata/data, file size, and etc. In our experiments, we modify two modern file systems in Linux, ext4 and btrfs to employ such kinds of access hints. The results show that efficient intelligent data placement is achieved by several file system level static hints.

- Stream ID: The stream ID is used to classify different data and make the associated ones storing together or closely located on one device. It helps storage systems to improve caching efficiency, space allocation and system reliability (in flash memory-based storage devices), etc. HintStor exposes an interface in user space so that applications can assign a stream ID for writes in each application as well as across applications. We run both a synthetic workload and a key-value database with different levels of journaling policies to evaluate stream ID as access hints.

- Cloud prefetch: In this case, we study the hints to efficiently integrate local storage with off-premise cloud storage. Data blocks are expected to automatically transfer from local servers to remote cloud storage. We simulate a heuristic cloud prefetch mechanism with pre-reserved local cache space. When data transfer from cloud to local, the latency may degrade the overall performance dramatically. We mimic different network bandwidths in the experiment. The results show that HintStor provides the ability to evaluate I/O access hints in a hybrid cloud environment.

The remaining of this chapter is organized as follows. In Chapter 2.2, we present the background of our work. Chapter 2.3 describes the design of HintStor. In Chapter 2.4 we demonstrate three different access hints with several storage configurations and show the evaluation results. Chapter 2.5 summarizes the related work. Finally, we conclude this chapter in Chapter 2.6.

2.2 Background

I/O access hints seem like a promising way of improving performance and easing management for future heterogeneous storage systems. In the previous studies of access hints [9, 31], caching and prefetching in the host machine are mainly considered. Some

off-the-shelf system calls in Linux, such as $posix\_fadvise()$ [32] and $ionice()$ [33], may be used for access hints. For example, $sys\_fadvise64\_64()$ [32] can specify the random access flag to the kernel so that the kernel can choose appropriate read-ahead and caching techniques to improve access speed to the corresponding file.

However, this flag in $sys\_fadvise64\_64()$ [32] is used in the Linux kernel and performed in the page cache level.Page cache usually refers to the storage of main memory. While in enterprise storage systems, storage itself may contain storage devices with different storage media like flash and SCM, which can be served as a portion of the storage volume. Unlike OS-level storage virtualization, to achieve better QoS for different applications in a hybrid or heterogeneous storage environment, intelligent data movement plays an essential role to make use of different types of storage devices. However, the existing prefetching engine in Linux does not support dynamic data movement in such a storage configuration. Thus, $fadvise()$ and $ionice()$ both used for read-ahead purpose are not applicable.

Some of the previous work concentrated on the classification of file data (file type, file size, metadata), which are called static hints in this chapter. For real-time data requests, the access patterns change from time to time. To make the storage system react quickly, it requires data movement across different devices. Thus, we need to design a framework that can evaluate a composited storage volume for persistence which is different from flashcache [34] and bcache [35]. After applying various I/O access hints, the underlying storage system performance can be improved via storage level data management like data migration.

Some existing file systems support customized classifications. For example, btrfs can support different volumes in the same file system [36]. However, btrfs asks the user to statically configure the volume for the storage. The host may have several applications. Different file systems have different storage requirements. The users may use one logical volume to support multiple applications. Thus, to achieve efficient data management, we need to consider dynamic access hints instead of the static information pre-configured in volume level.

To apply hints, without standard APIs, we may need to modify each application and even the whole OS. The current SCSI T-10 and T13 organization have proposed some standardized interfaces for access hints [30], but storage industry is still trying finding a

way to make all the stakeholders agreeing with these new standards/protocols. Our work is trying to design and implement a framework for both legacy and modern computing systems to design and evaluate the access hints from different layers. By means of slight modification of each layer, we may design/build a flexible framework to achieve this goal.

2.3 HintStor

2.3.1 Prerequisite

To manage and implement I/O access hints, the prerequisite is that we can manipulate data in the block storage layer. The modularized kernel allows inserting new device drivers into the kernel. Device Mapper (DM) [37] is an open source framework in almost all Linux systems to provide a composited volume with differentiated storage devices using various mapping polices (e.g. linear and mirror). We implement two generic block level mapping mechanisms as two new target drivers in DM (kernel version 3.13.0) as shown in Figure 2.1. These two mechanisms in data storage management can be used to response to and take advantages of enabled access hints. They in total contain ∼600 lines of C code in the kernel. The two targets named redirector and migrator are described below:

- **Redirector**: With the redirector target, the target device ($bio \rightarrow bdev$) can be reset to the desired device (e.g. /dev/sdd). A mapping table is used to record the entries of the devices for each request where the original address has been changed. As a result, the destination of the write requests associated with bio data structure can be reassigned to any appointed device as long as it belongs to the composited volume.

- **Migrator**: We implement the Migrator target driver in DM using the "kcopyd" policy in the kernel which provides the asynchronous function of copying a set of consecutive sectors from one block device to other block devices [38]. Data blocks in the migrator are grouped and divided into fixed-size chunks. Each time the data blocks in a chunk associated with the I/O requests can be moved from one device to another device. Besides, migrator can also carry out data replication

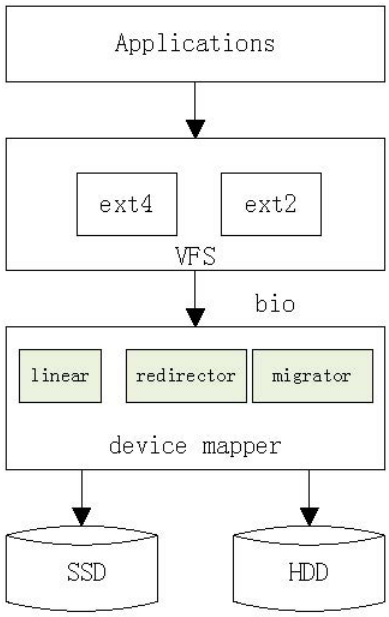


Figure 2.1: Block redirector and migrator targets in device mapper.

function. In the target driver interface, a flag REPLICA is used to mark whether keeping or removing the original copy pointers after the migration. The mapping table records the entry changes with each operation.

We modify dmsetup in LVM2 [39] to call the above two new drivers. Once the incoming data blocks arrive at the block level, the volume manager can choose either placing them on the original location or redirecting to a new location. Even after the data blocks have been placed, they can be migrated to a new location (different or the same device) or replicated with duplications. With the new functions in DM, we will show how we design HintStor to carry out I/O access hints in Chapter 2.3.2.

2.3.2 HintStor framework

In legacy Linux operating systems, each block driver registers itself in the kernel as a device file. Each request structure is managed by a linked list bio structure [40]. The core of bio, *bi_io_vec*, links multiple fixed-size pages (4KB). Developers can build logical device drivers with various mapping policies in DM. However, these drivers manipulate data blocks with small sizes and no data migration policy is implemented. We have

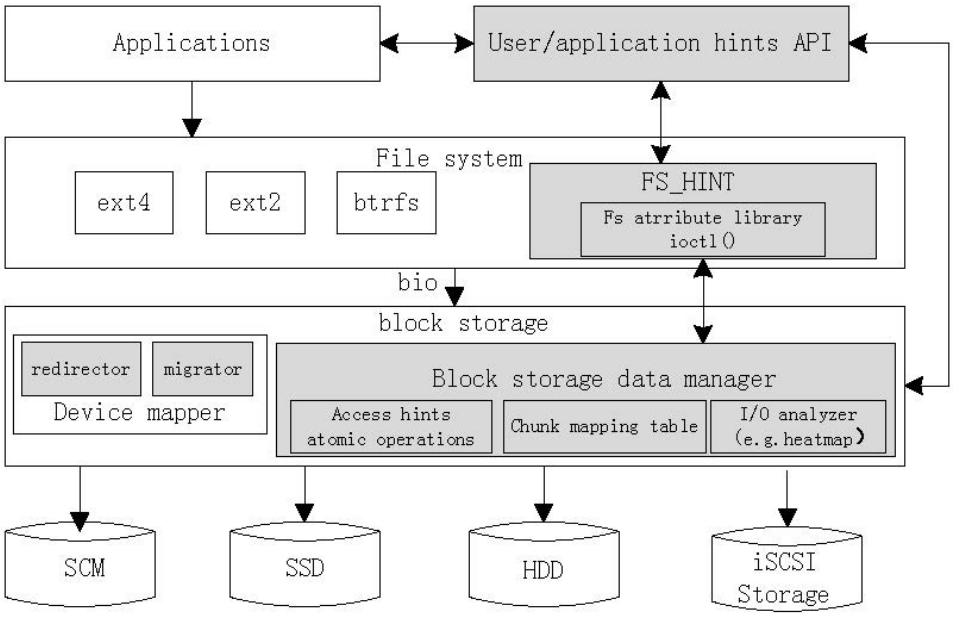


Figure 2.2: HintStor framework.

developed HintStor with the following new features: 1) To control data management of large size chunks and perform chunk level movement/migration, HintStor implements a new block driver in kernel. 2) To extract file system semantics of a chunk, HintStor hooks a plugin in file system level to exploit internal file data structure and the file data location. 3) To connect the applications and block storage manager, an interface is exposed in the user space for users and applications to exploit and receive information from both file systems and block storage manager. Moreover, this interface allows users/application to send access hints to storage layer. The new block driver is enabled to parse external access hints and execute certain commands. Figure 2.2 depicts the hierarchy of HintStor which is mainly composed of three levels including new interfaces and plugins in application level, file system level and block storage level (all the shaded components). In the following we will elaborate them from bottom to top.

1.) Block Storage Data Manager

HintStor treats data placement and data movement as essential functions in improving the performance of a heterogeneous storage system. To make the underlying block layer carry out data movement operations and perform access hints, we devise

and implement a block storage data manager extending from DM using the two basic target drivers described in Chapter 2.3.1. Block storage data manager mainly contains a chunk mapping table, a chunk-level I/O analyzer and a model of access hint atomic operations.

Chunk mapping table: Block storage maintains the Logical Block Address (LBA) to Physical Block Address (PBA) mapping. HintStor implements a block-level data management with the granularity of fixed-size chunks. This simulates the scenario of enterprise storage management [17]. The size of a chunk is usually much larger than 4KB. For example, HintStor configures 1 MB as the chunk size in our experiments.

Chunk-level I/O analyzer: HintStor monitors the I/O access statistics based on each chunk. To measure the access frequency of each chunk, a heatmap is used to represent the data access information in a period. In the user-level, a tool by Perl is developed to visualize the real-time access statistics.

Access hints atomic operations: One of the potential ways of passing access hints is using SCSI level commands based on T10 and T13 standard proposal [30, 41]. In [10], they associate each block I/O request a classifier in OS via using the 5-bit SCSI field. HintStor does not implement the real SCSI commands. Alternatively, HintStor emulates the SCSI based command through inserting new APIs to represent hints in each level. One reason of doing so is for a quick prototyping. For example, to use the new SCSI commands, we have to modify each driver like iSCSI target driver. The other reason is to make HintStor more flexible by avoiding laborious block driver modifications. Each access hint command is a four-item tuple as $(op, chunk\_id, src\_addr, dest\_addr)$, which contains the operation type, the chunk ID number, the source address and the destination address in the logical volume. The data management in block level mainly contains four fundamental atomic operations, REDIRECT, MIGRATE, REPLICATE and PREFETCH.

- **REDIRECT**: The REDIRECT operation happens when the real I/O has not finished but the length of the waiting list queue is larger than a predefined threshold. The redirection function mainly calls the redirector driver in DM. When the bio request comes into device mapper, REDIRECT will reassign the data chunk to the destination location. For example, in a composited logical volume with SSD and HDD, if the HDD is overloaded with many random writing I/O requests, data

manager can issue multiple REDIRECT operations to send data chunks to SSD instead of HDD.

- **MIGRATE**: Data migration plays a crucial role in HintStor. To enable data migration, a data migrator daemon is running on the background. When data blocks are originally placed in a storage with differentiated storage devices, the data access frequencies may change from time to time. The MIGRATE operation moves the data chunk from the original place to the destination place via call the migrator target driver in DM. To guarantee consistency, during the migration process, the chunk is locked regardless of the incoming requests on this chunk. HintStor provides two different types of data migration. One is triggered either by users or by applications. The other one is timer-based heuristic migration policies. For example, the user can configure the data migration every two hour to migrate the top-k frequently accessed chunks in the heatmap to the fastest device.

- **REPLICATE**: The replication function is used to keep replicas of a data chunk which is assigned by the applications. It is similar like the mirror target of the Linux software RAID, but we use large chunk size (1MB or bigger) to manage the replicas. HintStor makes use of the migrator target driver in DM by slightly modifying the return process. That is making the mapping table keep the original copy pointer. Many distributed data storage systems recommend the user to keep at least three copies of the data such like some big data processing platforms (e.g. HDFS[42], Kafka[43]). In a distributed storage environment, adding more replicas of the hot chunks across multiple devices will improve data availability and reduce the mean response time. We can use REPICATE to create multiple duplications to emulate such cases. Similarto MIGRATE, the REPLICATE operation locks the chunk during the execution process.

- **PREFETCH**: The PREFETCH operation is similar to buffering. In the initial configuration, HintStor supports reserving a portion of space for the prefetching buffer. The PREFETCH operation will load data chunks from the original space to the buffer space. The implementation of PREFETCH is similar to REPLICATE and MIGRATE. The major difference is that PREFETCH does not need to finish

copying the chunk before serving new incoming I/O to this chunk.

2.) File System Plugins

File systems usually encapsulate the internal data management of files, to provide a few POSIX APIs such as read(), stat()[44]. Each filesystem is implemented in its own way [44]. A filesystem usually consists of metadata blocks and data blocks. The metadata information includes block bitmap, inode, superblock, and so on. When the block level storage is enabled to execute chunk level data operation, HintStor first considers using access hints from filesystems. To improve filesystem performance, the metadata location is expected on a faster storage device since this information is accessed more frequently. To help users classify filesystem internal data blocks, HintStor provides a plugin in VFS level, called FS_HINT. In FS_HINT, a filesystem attribute library including different dimensions like file size, metadata/data, is pre-configured. To date, FS_HINT supports the mainstream filesystems in current Linux, like ext2, ext4 and btrfs. For example, ext4 allocates a block for metadata via $ext4\_new\_meta\_blocks()$ according to the multiblock allocation mechanism, and FS_HINT captures the returned block group number ($ext4\_fsblk\_t$). The information collected from this library is exposed to both user-space and applications. Since we design HintStor as a flexible framework, it allows the researchers to enrich new features and use the basic components for I/O access hints study. Thus, FS_HINT is open for adding more attributes.

Continuing, a filesystem usually does not have the knowledge of the underlying block storage component (e.g. a hybrid storage logical volume with SSD and HDD). Thus, file systems generally do not manage the data across different storage devices by themselves. Btrfs allows users to configure a sub-volume with a certain block device or logical device for special use cases. However, btrfs does not deal with data block placement across different volumes. To control the block allocation, ext2 uses $ext2\_new\_blocks()$ (balloc.c) to allocate new blocks from the block group. However, for ext4 and btrfs, the implementation is much more complicated when the plugin involves kernel code modification. In Linux, the file ioctl() interface manipulates the block device parameters of special files based on the file descriptor. The users and applications can query the approximate file boundary in a logical volume via ioctl() interface. We use the fiemap ioctl() method from user-space to retrieve the file extent mappings [45]. Most filesystems like ext4 and btrfs support extent. Without block-to-block mapping, fiemap returns the associated

extents to a file. To make the underlying block storage understand the physical location of a file, we expose this interface to applications to map the logical block address and the addresses inside the logical volume. The returned locations of each extent are sent to the block level.

As HintStor manages the block level data in chunk level, the chunk to extent mapping may be unaligned. Depending on the size of the file, HintStor will prioritize the extents from small size files to characterize the chunks. HintStor mainly studies the data management in storage level with multiple storage devices, each of which has large capacity compared with OS cache study. Thus, even we sacrifice a little accuracy of a small portion of data blocks, the results are still closely representing the real cases. As a result, FS_HINT may not contain all the accurate mappings of each file but this does not affect the I/O access hint evaluation. This approach helps us quickly prototype HintStor. In addition, FS_HINT is compatible on most of Linux platforms. We make the LBA to PBA mapping in a file as a function in FS_HINT. The major function contains only about ∼50 LoC. Furthermore, FS_HINT supports querying the mappings for multiple files without complex kernel code modification.

3.) Application/User Level Interface

In the traditional Linux OS, applications generally do not directly send extra information to the back-end storage controllers. Applications and users access data via interfaces like file or object in the past decades. To accept and parse user defined access hints, HintStor makes use of the sysfs interface [46] to build an attribute library for applications to communicate with block storage, file system and block storage. sysfs works as a pseudo file system in Linux kernel which exports brief message from different kernel subsystems. In HintStor, we add the kernel part in the block level as shown in Figure 2.2. Then, the predefined commands from user space can be triggered and performed in the kernel. For example, if the user makes the MIGRATE request, the command will be passed from the user space to the kernel. In the kernel, the migrator daemon will trigger the LBA to PBA retrieval process and finally block storage will be informed which data chunks and where they are going to be migrated. HintStor calculates the ratio of different types of storage. For an application, if a file is created, the application can selectively apply the hints to this file. We have a hints-wakeup program to ask the application to confirm this information. For example, if the file is supposed to be a hot

file in a period, HintStor will try to load this file into a fast storage. For a period of time, this file is not used, and also the application may be unsure how long it will keep hot. The block storage will connect the application to see if this file is not going to be used. In the contrast, if a file is initially a cold file, but the application may not be aware of this, in the block storage, the block storage will wake up the hints-generator to produce a hint to the block storage. In the current version, HintStor supports basic Linux operations like cp, dd, fsck, redis key-value database, benchmark tools (e.g. fio [47] and filebench [48]). In addition, HintStor supports user defined APIs such like stream ID and cloud prefetch. We will demonstrate some use cases in Chapter 2.4.

4.) I/O Access Hint Classification in HintStor

Basically, I/O access hints can be either statically extracted from the existing systems or dynamic captured in the systems and even added by the users/applications. Table 2.1 shows the two categories of access hints and some examples. The filesystem plugin allows the system designers to send the filesystem information to block storage and applications. The information includes the metadata/data, file size, etc. HintStor can decide the initial location of different types of data, according the static access hints. To achieve better QoS for the applications, intelligent data movement plays an essential role to make use of the large low cost storage space (e.g. cloud storage and HDD) and also avoid the longer network latency. HintStor is designed to further study data migration, space relocation and prefetch operation controlled by the I/O access hints. Dynamic access hints are aiming to help block storage manage data placement in the running time. For a cold file is opened again, such system call will be captured to help block storage make prefetching decision. Dynamic I/O access hints can be obtained through analyzing the workload or assisted by the applications and users during the running time. Such access hints go across different layers in the host OS from user space to kernel space using the user defined APIs. HintStor leaves this design open and makes it flexible for developers. In Chapter 2.4, we will show how we use HintStor to carry out some of them and evaluate their effectiveness on storage systems.

Table 2.1: I/O access hints categories and some examples.

I/O access hints classification	Examples
Static I/O access hints	metadata/data file size file type
Dynamic I/O access hints	open a file write/read a file stream ID cloud prefetch

2.4 Evaluation

In this subchapter, we will show how we perform and evaluate I/O access hints in HintStor with three types of I/O access hints.

2.4.1 File System Data Classification

In this evaluation, we will show how HintStor plays with file level static hints. We assume a hybrid storage system with a faster device and a slower device. To study file system access hints, we define a two-level file data classification model as shown in Table 2.2. Block storage layer manager can make data placement decisions based on the data classification of each chunk. Based on level-1-hints (metadata or data), the block storage data manager will place the chunks of metadata info on a faster device. In most cases, HintStor will reserve some space on faster device for metadata related chunks. The data blocks are placed on the other slower device. For the cases of level-2-hints, data manager not only places the blocks with file metadata information on faster device using Level-1-hint but also further selects the chunks from the smallest size files to place on the fastest device. For example, if the underlying devices are SSD and HDD, the data manager can select the file size smaller than a certain value (e.g. 1MB) to store on SSD. We change the *sys_read*() system call to monitor file operations. HintStor will take actions to call the REDIRECT function if the corresponding data chunks are not allocated to the desired location. For the No-hint case, the composed volume is configured using the linear target driver in Device Mapper. FS_HINT in HintStor is disabled in the No-hint cases.

Table 2.2: Two-level file system data classification example.

Level-1-hint	metadata (e.g. inode, bitmap, superblock), data blocks
Level-2-hint	File size range ([0,256KB), [256KB,1MB), [1MB,10MB), [10MB,100MB), [100MB,1GB), [1GB, ∞)

We use filebench [48] to create a set of files in this hybrid storage. The file sever contains 10248 files in 20 directories. We create 48 files with 1GB size and 200 files with 100MB size. The sizes of the rest files are uniformly distributed among 1KB, 4KB, 6KB, 64KB, 256KB, 1MB, 4MB, 10MB and 50MB. The total capacity of the files is about 126 GB. The workload is similar to the Fileserver workload in Filebench. Chunk size is set to 1 MB. We run each test for 500 seconds. Each time Filebench runs the same number of operations. In addition, each test is running 3 times and we take the average value as the result. In the end of each test, Linux will trigger the flush() and sync() command to clear the cached pages.

We first test a storage configuration with two different storage devices, HDD (120GB) and SSD (40GB). With Level-1-hints, all the chunks with metadata will be placed on SSD. With Level-2-hints, HintStor chooses the file size smaller than 10MB to store on SSD. We compare the results with three mechanisms including no-hint, Level-1 hints and level-2 hints on ext4 and btrfs. As shown in Figure 2.3, the average I/O latency in both btrfs and ext4 is reduced by adding file system data classification hints. Compared with the No-hint cases, by adding Level-1-hints, the latency on ext4 and btrfs is reduced by 48.6% and 50.1%, respectively. By further adding Level-2-hints, the latency is reduced by 12.4% and 14.8% compared with the cases only using Level-1-hints.

To demonstrate the flexibility with different types of devices in HintStor, we build a heterogeneous storage with HDD (120GB), SSD (35GB) and SCM (5GB). We use a fixed DRAM space (10GB) to emulate an SCM device. HintStor reserves 2 GB space on SCM to keep metadata chunks. Based on Level-1-hints, the data manager will first place the data blocks on SCM unless SCM is out of space and then data will go to SSD. Based on Level-2-hints, files with sizes smaller than 256KB will be placed on SCM. SSD is used to keep files with sizes from 256KB to 100MB. The files with sizes larger than 100MB will be placed on HDD. As shown in Figure 2.4, by adding Level-1-hints, the average request latency is reduced by 42.1% and 43.3%, compared with the No-hint

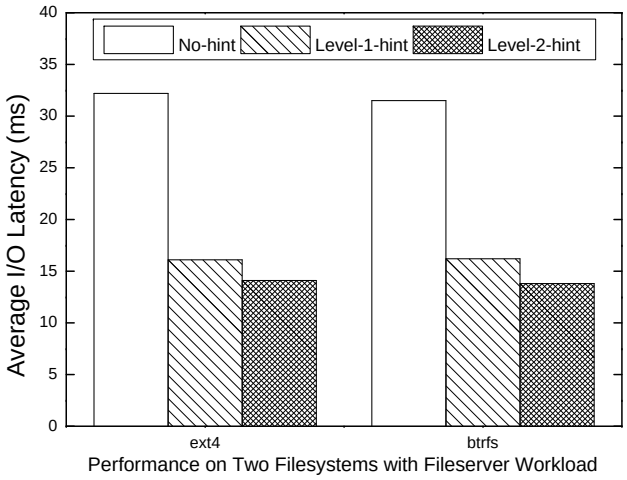


Figure 2.3: The average latency on a hybrid SSD and HDD volume.

cases. With Level-2-hints, the average I/O latency is reduced by 40.4% and 46.1%, compared with the Level-1-hint cases.

The previous study on access hints were focusing on cache improvement, such as SSD caching. HintStor can evaluate the effect of access hints on data movement across different storage devices. In the next experiment, we show the combination of data migration. As presented in Chapter 2.3.2, HintStor can perform data migration inside the virtual disk by calling the MIGRATE function. We configure a heuristic data migration policy by moving the Top-1000 frequently accessed chunks into the SCM in a fixed period. We can configure various migration intervals to investigate how data migration affects the performance. HintStor calculates the accumulated statistics at the end of each interval and triggers data migration if it is needed. Figure 2.5 and Figure 2.6 show the average latency for ext4 and btrfs when data migration in the block level is triggered in different intervals. We run the test for two hours. As shown in Figure 2.5 and Figure 2.6, the average I/O latencies in both ext4 and btrfs are reduced by adding access hints. Compared with the above case, there is more improvement in this case where the storage system provides data migration management. Without hints, both performance can be improved with moving frequently accessed data into the fastest

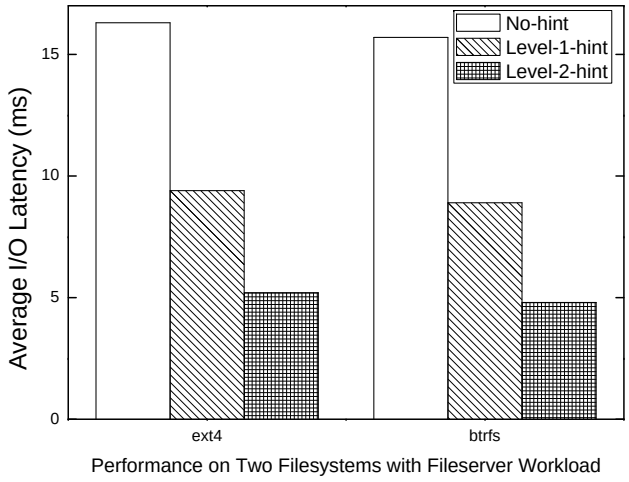


Figure 2.4: The average latency on a hybrid SCM, SSD and HDD volume.

storage. However, block level storage does not know the internal file data distribution. By recognizing the file data structure, the initial data placement can be improved via Level-1-hint. Through further exploiting Level-2-hint, the small size files are placed on the faster pool. Both reduce the data migration load.

Each *sys_read*() operation will trigger the execution process of access hints. The computation complexity related to access hints generation is O(1). To measure the resources used by access hints, we calculation the average CPU utilization for the above cases. The average cpu utilization caused by Level-1-hint and Level-2-hint in all the cases is less than 2%. Thus, access hints operations are lightweight in HintStor.

From the experiments in Chapter 2.4.1, we evaluate access hints from two file systems by exploiting their internal data structure and attributes in HintStor. To demonstrate its flexibility, we configure the storage system with different types of devices. Differing from the previous study [9, 10], HintStor can evaluate access hints with various block level data migration polices. Moreover, access hints execution in storage system brings very slight computation overhead.

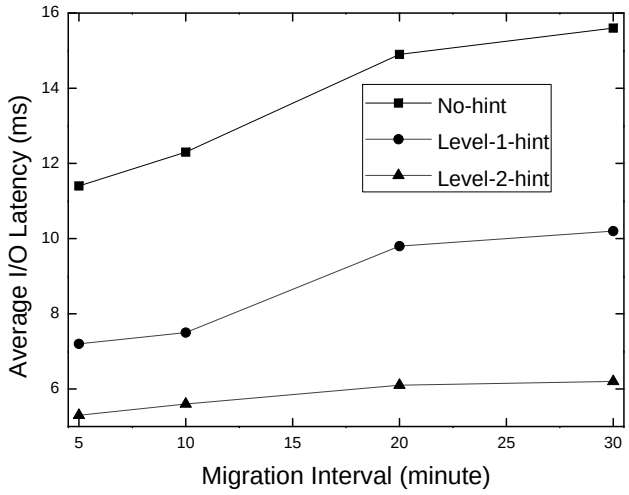


Figure 2.5: The average latency with different migration intervals for ext4.

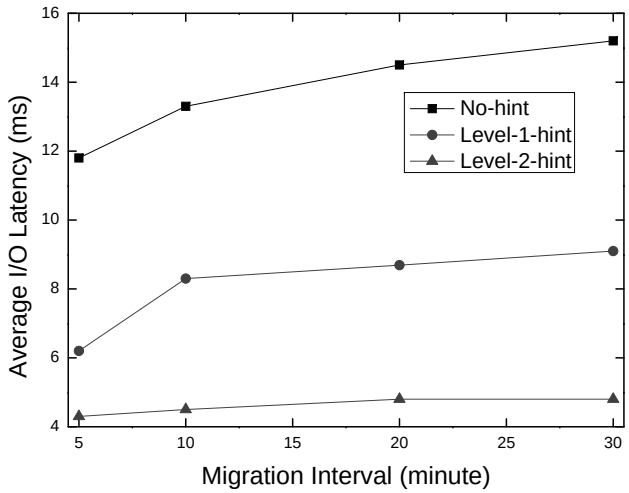


Figure 2.6: The average latency with different migration intervals for btrfs.

2.4.2 Stream ID

Employing application-level data classification usually requires modifications of each application. We take the idea of stream ID [49] as an example of the application level access hints. Stream ID is used to aggregate the associated data into one stream and assist the storage controller to make proper data placement. We use a mapping table in HintStor to record all the stream ID numbers.

There are several ways of employing stream IDs in the current I/O stack. One way is using the existing Linux POSIX APIs like *posix_advise*(). I/O optimization is achieved by means of providing prefetching and caching recommendations to Linux kernel. If the system wants to distinguish the data based on a file system, then it needs to modify each file system. However, the application may be designed to run on different file systems. In this case, simply exposing an interface in the user space for applications in HintStor may be better than changing the file systems. The applications just inform the devices the access pattern of the I/O requests so that the devices can allocate space based on each stream (files).

HintStor exposes a user/application interface in the user space to let users define their own access hints with few modifications. Although the API implementation includes both functions in user space and kernel space, for general API like file name and access patterns, they are implemented in kernel and do not need further kernel level modification. Here, we configure the stream ID for write requests. The storage manager layer can allocate appropriate storage space for different streams by taking the information of stream identifiers. In our implementation, we define and design an enhanced stream ID hints including two fields as shown in Figure 2.7. The first field indicates the I/O access pattern. The application can specify an access pattern descriptor based on the pattern list (e.g. write intensive, read intensive, archival/backup data). The application can either set the first field or leaves it as default (unknown). The second field is the tag which records the stream ID number. We use both synthetic workload and a real key-value database to evaluate stream ID.

Case 1: Synthetic Workload. We modified the $FB\_open()$ interface in filebench [48] to execute the user level access hint API. HintStor takes the file descriptor to construct the stream ID when $FB\_open()$ is called. As in the beginning there is no clue of the access pattern of each file, the access pattern field is configured as ünknown". Note this

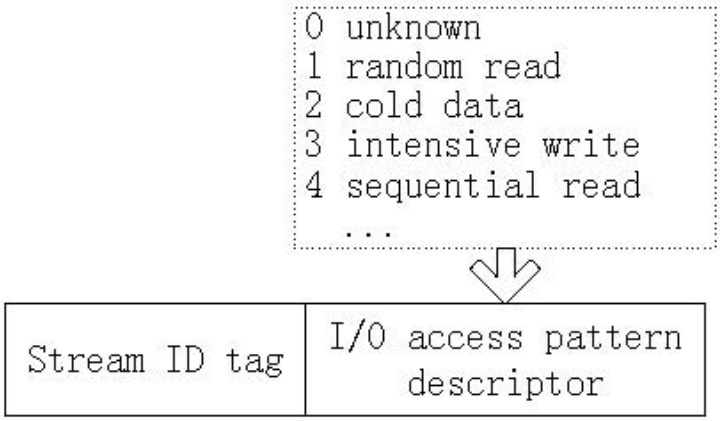


Figure 2.7: Diagram of the enhanced stream ID.

does not require any further file system modification. We create multiple data streams in the user level, each of which has different properties. For all the streams created by the I/O generator, each will be given a stream ID number. Each time filebench creates a new file, a stream ID will be generated and passed to HintStor. In this test environment, we create four types of files. Each type of files is tested using one of the four predefined workloads: Webserver, Fileserver, Singlestreamwrite (iosize is set to 20GB) and Randomwrite. In total, we create 1560 files, with total capacity of 125GB. The hybrid volume is configured with 150 GB using 20 GB SSD, 125GB HDD and 5GB SCM. The access hints associated with the data blocks will be analyzed by the data manager in block level. Data manager will look up the hints as well as storage pool free space. For initial data placement, data blocks will be placed based on the stream ID hints. To leverage the properties of each device, the data manager will place the Singlestreamwrite data on HDD, Webserver data on SSD, Randomwrite data on SCM. The data associated the fileserver workload will be place either on SSD or SCM, depending on the remaining capacity of each storage device. Internally, HintStor will migrate the data chunks based on the heuristic migration policy described in Chapter 2.3. Random write pattern is prioritized to reside on SCM. SSD is more appropriate for random read data. HDD is serving sequential patterns and as well as cold data.

The migration timer is set to happen every 10 minutes. We run the test for 2 hours. We measure the system throughput with stream ID and without, respectively. As shown

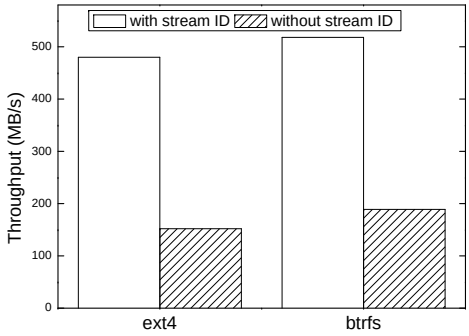


Figure 2.8: System throughput improvement with stream ID.

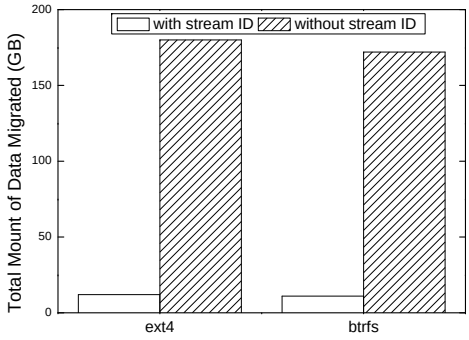


Figure 2.9: The total amount of data migrated in HintStor with Stream ID and without.

in Figure 2.8, system throughput in the case with stream ID outperforms the result of no stream ID case by 3.2X and 2.7X in ext4 and btrfs, respectively. Figure 2.9 shows the total amount data migrated in these two cases. For ext4, adding stream ID, the total data during migration is reduced by 93.5% and 92.4% for ext4 and btrfs. With the enhanced stream ID model, the storage manager can make better initial storage allocation decisions for different application data. Thus, the migration cost is alleviated dramatically. In addition, migration traffic affects the incoming I/O requests and increases the response time.

The overhead of adding the stream ID information is similar as the modification

of *sys_open*(). Hence, there is little impact on the system performance by performing access hints in filebench.

Case 2: redis database. redis is a popular in-memory key value store [50]. Although redis is running in memory, it provides two persistency policies [51]. One is called RDB persistence which performs point-in-time snapshots of the dataset at certain intervals (e.g. every 24 hours). The other is AOF which can log every write with different fsync policies: no fsync, fsync every second, fsync at every query, etc.

In HintStor, we add stream IDs for redis database. To run multiple redis instances in a composited logical volume efficiently, there are multiple AOF and RDB files. To use the Stream ID API in HintStor, each AOF and RDB file in each instance is configured with a number as a tag. The AOF files are tagged using the access pattern descriptor "frequently write". The RDB files use the "cold data" access pattern descriptor. Based on the stream ID, block storage data manager can allocate the low speed storage to keep the large size backup file which is used by RDB, but use the high-speed storage to store the short-term snapshot files for AOF. If the volume contains a portion of storage from the off-premise cloud, the short-term files do not need to be transferred to cloud.

We configure the volume with a local SSD with 50GB capacity and a 200GB remote iSCSI HDD storage. Ext4 is running on top of the volume. We use Workload A (update-heavy) from the YCSB framework suite [52]. We simulated 50 clients running on a single machine. This machine generates about 10 million operations per second. Both AOF and RDP are enabled. In total, we run 6 redis instances on an 8-core server with 48 GB memory. Different fync frequencies of logging and journaling mechanism in redis's persistence configuration achieve different levels of reliability. However, frequently committing message to storage leads to system performance degradation. The fsync policies are set to be per query, per second and every 30 seconds. As shown in Figure 2.10, the throughput in the case with stream ID outperforms the one without stream ID by 475%, 365% and 204% for the three fsync policies. Workload A mostly consists of write requests which also produce many logs for AOF. Thereby, if the AOF file is on the high latency storage device, the performance will be very poor. Thus, the per-query fsync policy without stream ID has very low performance. Adding stream IDs, redis with different levels of persistence can achieve much better performance by efficiently utilizing the underlying heterogeneous storage devices.

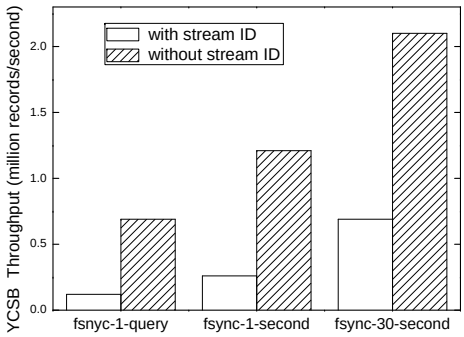


Figure 2.10: YCSB throughput for redis with three fsnyc polices.

In Chapter 2.4.2, we run two cases and show the ability of HintStor to implement and evaluate the stream ID access hints based on the user/application level interface. System designers can define and configure their own access hints in user space even without touching the kernel.

2.4.3 Cloud Prefetch

Cloud storage emerges as a mainstream for reducing cost and increasing data reliability. We can build storage services which is based on the integration of on-premise local storage and off-premise cloud storage. Such hybrid cloud is one of the appealing solutions to embrace cloud. The storage interface provided from a hybrid storage can either be a high level file interface or a block interface. In the following, we will show an example to build a block level volume in a simple hybrid cloud environment with HintStor framework to evaluate potential access hints.

Cloud infrastructure is always deployed in the remote site and thus, network latency is one of the major bottlenecks for leveraging cloud storage. To preload the data into local servers, one of the strategies is explicitly sending prefetching hints to the cloud. We define and implement cloud prefetch in HintStor to evaluate storage management in the cloud environment. With the user/application level interface, applications can appoint which files to buffer either on local side or cloud side. Cloud prefetch calls the PREFETCH operation in HintStor to fetch the data blocks to the local devices.

HintStor supports reserving a portion of the local storage as a buffer in the volume. This prefetching space for cloud storage can be released when the storage space is close to use up.

In the experiment, we configure the local storage with 10 GB HDD and 1GB SSD. HintStor currently does not support S3 interface. We use the iSCSI based storage with one remote HDD to emulate the cloud storage. The capacity for the iSCSI target is 40 GB. The local and cloud storage is configured as a single volume on top of which ext4 file system is mounted. We use the Linux traffic control tool *tc* [53] to mimic the network latency.

We use *dd* (generated from /dev/zero) to create two 1GB files and 100 1MB files in a single directory. Without access hints, the ext4 filesystem has no clue to place the data and will allocate free extents for these two 1 GB files. With the file-level data classification hints, the metadata will be place on the local storage (on SSD), the small size files are placed on the local HDD while the two large size files will be placed in the cloud. During the file creation process, a portion of the local storage from HDD is used for cloud prefetching based on the network latency. We instrument a set of user level access hints with the dd command. In this test, the data manager reserves 1 GB local storage buffer for 1Gbps connection and 2 GB local storage buffer for 200Mbps connection. For 1Gbps network, when we start to create the first 1 GB file, a cloud prefetch hint is send to block level. While for the 200 Mbps network, cloud prefetch is triggered along with the creations of both 1 GB files.

Then, we use *cp* to copy the whole directory into the local /tmp directory and in the meantime the prefetching thread is triggered to prefetch the data blocks from the cloud to the local storage. We test the total elapsed time of copying all the 102 files in three access hints configurations, no-hint, Level-2-hint (described in Chapter 2.4.1) and cloud-prefetch-hint. Figure 2.11 shows the total execution time of each case. In the case of 1Gbps network connection, compared with the no-hint case, the execution time of using Level-2-hint and cloud-prefetch-hint is reduced by 13% and 66%. For 200Mbps connection, the total elapsed time is decreased by 34% and 71% with these two access hints.

We summarize the normalized total read I/O amount from cloud storage in Figure 2.12. Assisted by the cloud prefetch, the file data is buffered on the local storage. As

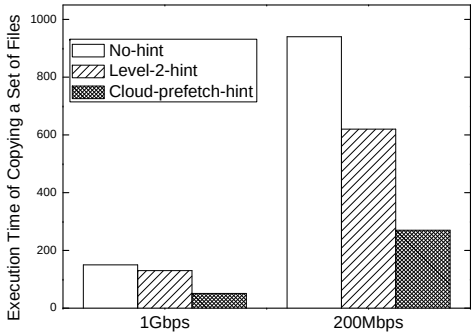


Figure 2.11: Execution time of *cp* a set of files which are placed across local and cloud storage.

we issue user defined cloud prefetch according to different network speeds and buffer sizes, the total data read from cloud in the 200 Mbps case is less than that of the 1 Gbps case. As the network latency increases, the cloud prefetch hint plus user defined access hints can improve more of read performance.

In the above cloud prefetch example, access hints are performed like the file system data classification. Thus, the computation overhead is very low for access hints execution. We use a small portion of the local storage as a buffer to prefetch the data blocks. If there is no free space in local storage after the storage system is fully used, cloud prefetch hint will be disabled.

Concluding from this example, with HintStor, we can emulate a storage system with local and off-premise cloud storage. The access latency from network can be simulated. We can configure the internal buffer space of the volume to play with different user defined cloud prefetch hints.

2.5 Related work

As aforementioned, more new storage technologies are emerging and (e.g. SMR [2, 54], SCM [6, 55]) making the storage systems more heterogeneous. Each storage media has their own idiosyncrasies with different cost and performance. To efficiently make use of hybrid storage devices in a storage system, the legacy way is building a tired

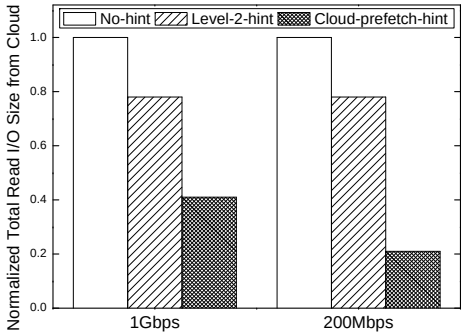


Figure 2.12: Normalized total read I/O size from cloud storage.

storage system [13, 17]. Such systems always make data tiering via data movement via monitoring I/O in block/chunk level. The limitation of legacy data tiering is lack of sufficient information to distinguish block boundaries from different applications so as not being able to place relative data blocks together.

Researchers started to pay attention to introduce access hints into storage systems by enhancing caching and prefetching [9, 10, 31]. Linux uses readahead system call or related mechanisms to help the OS to adapt the conventional hard drives, in general, to do conservative sequential prefetching. Aggressive prefetching is an effective technique for reducing the running time of disk-bound applications. Through speculatively pre-execute the code in an application, the OS may exploit some hints for its future read requests [31]. Even host-side cache is considered to improve write access to network based storage [56]. To achieve full benefit of Non-volatile write cache, a request-oriented admission policy associated with critical processes in the OS to represent upper level applications is proposed to cache writes awaited when the request is the process of execution [57]. As mentioned in the beginning of this chapter, Sonam Mandal [9] studied block-layer data deduplication by using hints. OneStore tries to integrate local and cloud storage with application level access hints [58].

In addition, most existing access hint research usually focuses on static access hints. They mainly study using hints extracted from the existing components, like file system [10] or database [59], to assist OS prefetching and caching (external caching disks). Some

researchers have studied static SCSI-level hints for SSD caching in [10]. In [59], they exploited how the database level can modify the existing system to perform the hints commands. While data accesses change from time to time, the underlying system should move data across different devices adaptively. If system designers and applications can input dynamic information to the storage systems, this will help data blocks stay at the appropriate layer combining the tradeoff between cost and performance.

Du et.al. implemented T10 based object-based storage [60] which tries to combine both block and file interfaces. Recently, there are standardization efforts in T10 [30, 61] and T13 [41] by using a portion of SCSI command descriptor block (CDB) to execute I/O hints. Such new block I/O commands involve numerous efforts of I/O stack modification. Choi from Samsung suggested exposing a new interface in user space so that applications can directly design and configure the stream ID for their writes [49]. To date, we have not seen an open source framework which can provide such ability for users and applications to employ access hints. The major contribution in this work is designing and implementing a flexible framework to evaluate I/O access hints from various levels.

2.6 Conclusion

Putting hints on existing storage systems looks fancy but maybe painful for system designers. This chapter presents a generic and flexible framework, called HintStor, to quickly play with a set of access hints and evaluate their impact in heterogeneous storage systems. The design of HintStor contains a new application/user level interface, a file system plugin and a block storage data manager. With HintStor, storage systems composed of various storage devices can perform pre-devised data placement, space reallocation and data migration polices assisted by the added access hints. HintStor supports hints either statically extracted from the existing components (e.g. internal file system data structure) or defined and configured by the users (e.g. streaming classification). In the experiments, we evaluate three types of access hints: file system data classification, stream ID and cloud prefetch in a Linux platform. The results demonstrate that HintStor is able to execute and evaluate various I/O access hints under different scenarios at a low cost of reshaping the kernel and applications. Our HintStor is open-sourced and available for download at: `https://github.com/umn-cris/accessHints`. In the

future, we are planning to enrich the access hints library and as well as support more platforms (e.g. containers) and protocols (e.g S3).

Chapter 3

Workload-Aware Data Management across Differentiated Storage Pools

Moving data tier by tier may not be efficient and even worse it may lead to unnecessary data movements. In this chapter, we study the storage architecture with fully connected (i.e., data can move from one device to any other device instead of moving tier by tier) differential pools (each pool consists of storage devices of a particular type) to suit diverse types of workloads. To explore the internal access patterns and thus efficiently place data in such a fully connected topology, we propose a ***Ch***unk-level storag***e***-aware ***w***orkload ***Analyzer*** framework, simplified as ChewAnalyzer.

3.1 Introduction

Internet applications and cloud computing generate a huge amount of data, driving the emergence of large capacity high performance storage systems. To reduce the total cost of ownership (TCO), enterprise storage is seeking a way to efficiently incorporate different storage technologies/devices, each of which forms as a storage pool, into one storage system. The architecture of combining flash-based Solid-Stated Drives (SSD) [12] and Hard Disk Drives (HDD) has been extensively studied [13]. Today, there are

more emerging storage devices/ technologies like phase change-memory (PCM), spin-torque transfer RAM (STT-RAM) and memristors [4] that not only offer non-volatile memory but also promise high performance non-volatile storage [6]] (termed Storage Class Memory (SCM)). The performance of SCM is close to memory. However, the price of SCM, even exceeds that of a many-core CPU [6]. In the foreseeable future, the cost per bit of SCM is still much more expensive than the traditional storage like flash-based or spinning disks. Different devices/technologies may have their own unique performance characteristics. Such diverse storage technologies make enterprise storage hierarchies more interesting and diverse. What used to be two-tiered (dynamic random access memory (DRAM) and HDD) is quickly broadening into multiple tiers.

In the past, storage tiering has been viewed as a method of getting both performance and affordability. The expected process of data placement is that fast tiers serve a group of intensive workloads for better performance while slow tiers are persistently storing the rest of data blocks to provide lower cost [15]. Data migration happens tier by bier when data access frequency changes. However, moving data tier by tier may not be efficient and even worse it may lead to unnecessary data movements. Thus, with multiple storage pools, storage tiering needs not be either linear or side by side if diverse storage technologies/devices are used. With proper initial data placement and efficient data migration, storage systems may consist of multiple storage pools that are fully connected with data to be stored at a tier to suit its current workload profile. Please note that workload profile may change from time to time. Nevertheless, it is challenging to decide how data to be placed in a storage pool and when to migrate data from one storage pool to another. Each storage pool has its own unique price-performance tradeoffs and idiosyncrasies with respect to workload characteristics they support the best.

To control data movements, data are managed either at file-level or at block-level. In file-level, data movements between storage tiers are usually made transparent to upper-level applications through a Hierarchical Storage Management (HSM) software [62, 63]. In block-level, data blocks are divided into fixed-size or variable-size data chunks (or called extent) [17]. Compared with file-level management, block-level management is more transparent but challenging due to its limited access information. Current commercial multi-tier storage systems from Dell-EMC [64, 65], IBM [66], HPE [67] and

NetApp [68] provide a block-level interface to customers. In this chapter we focus on dealing with data blocks in chunk-level, which has been employed in the fields of Storage Area Network (SAN), Virtual Machine (VM) storage management, etc.

Accurate chunk level workload characterization can help the system understand what resources are adequate for the associated requests. The classical hot and cold data classification methodology is employed in the tier-by-tier case [16]. Each data chunk is characterized as certain pattern, typically, high or low IOPS in a period, and followed by a greedy migration policy to be moved between different storage pools [17]. However, employing different dimensions and granularities may generate entirely different access patterns. We investigate a set of enterprise block I/O workloads using different chunk sizes and different taxonomy rules. Typically, smaller chunk size incurs more metadata management while larger chunk size reduces the flexibility of data management. Our experimental analysis shows that chunk sizes play an essential role in a fast but small capacity device. Different taxonomy rules may partition data into totally different categories. Through our experiments in Chapter 3.2, it can be seen that they do impact greatly in the accuracy of chunk-level workload profiling. Moreover, workload profiling is highly related to the device technology. For example, it is valuable to pay more attention to write requests when both SSDs and SCM are available. Although both offer better random I/Os per second (IOPS) than HDDs, the not-in-place-update and lifetime issue still drive researchers and developers to reduce and even eliminate random write I/Os on flash based SSDs [18, 19]. Furthermore, applications like big data processing (e.g. Hadoop) have their own characters (e.g. streaming or batch processing) [17]. As a result, conventional storage workload analysis methods oriented to tier by tier cases are not applicable on multiple differentiated storage pools.

In this chapter, to explore data access patterns in such a fully connected topology, we propose a ***Ch***unk-level storag***e***-aware ***w***orkload ***Analyzer*** framework, simplified as ChewAnalyzer. Our trace-based experimental observations and findings motivate the design of ChewAnalyer, which leverages different storage techniques and workload transformations to conduct cost-effective data placement. The chunk pattern is characterized as the accesses in the associated data blocks in a chunk. ChewAnalyzer works like a middleware to exploit and detect both coarse- and fine- grained I/O access patterns. Then, it performs the cross-matching to place the associated data chunks into

the appropriate storage media. Moreover, the overall cost, capacity and performance are considered in the design.

Specifically, in this chapter we make the following contributions:

- We employ a detailed experimental study of several enterprise block I/O traces and show the shortcomings of the existing methods of profiling block level workloads, and as well as argue the limitations of managing data tier by tier.
- To reduce system overhead but guarantee profiling accuracy, we propose the ChewAnalyzer framework to conduct cost-effective data management across multiple differentiated storage pools. We use a Hierarchical Classifier to gradually perform flexible access pattern classification for each data chunk according to different taxonomy rules. In each classification step, the Chunk Placement Recommender looks up the pools status and Pattern-to-Pool chunk placement library to check if it can advise better chunk placement. Otherwise, the classification and detection process shifts to the next hierarchy.
- ChewAnalyzer++ is enhanced to make ChewAnalyzer achieve more accurate profiling through selecting a portion of data chunks to split them into small ones and then characterize them to better utilize the resources.
- We build a prototype equipped with Storage Class Memory (SCM), Solid State Drive (SSD) and Hard Disk Drive (HDD). We use trace-driven approach to evaluate our design in a Linux platform. Our experimental results show ChewAnalyzer outperforms the conventional dynamical tiering [17] with less latency and less write times in the flash pool. ChewAnalyzer++ can further reduce latency. The total data being migrated are also reduced.

3.2 I/O Workload Analysis and Motivations

In this subchapter, we investigate the access patterns of different enterprise block I/O traces. Differing from the workload analysis for caching, we focus on data access characteristics over longer durations. Page level cache replacement policies are usually making quick and heuristic decisions based on the temporal changes to impel memory. The decisions of data movements in large scale heterogeneous storage systems are more cautiously

Table 3.1: Storage I/O workload characterization dimensions.

I/O size
I/O Intensity
Read/Write ratio
Randomness/Sequentiality
Local-Access

Table 3.2: Chunk access pattern classification examples.

Taxonomy rule 1	Non-access	Inactive	Active
Taxonomy rule 2	Sequential	Random	
Taxonomy rule 3	Sequential	Random Read	Random Write
Taxonomy rule 4	Sequential write	Fully random write	Overwritten

and generally performing at predetermined intervals (by hour or by day) or on-demand. For the on-demand data migration, as long as the new placement has been completed, the status is expected to be persistent for a long period of time.

3.2.1 Preliminary of I/O workload analysis

To analyze the I/O traces, we divide the whole storage space into fixed-size or variable-size chunks. Each chunk contains a set of consecutive data blocks. Then, we characterize the collective I/O accesses to each chunk in a constant epoch (time window). The major dimensions we used are summarized in Table 3.1. I/O intensity means the average number of accesses in a time window. Read and write ratios are used to classify read or write dominated patterns. We study I/O Randomness/Sequentiality through different sequential detection policies [69]. High Local-Access ratio [70] describes the scenario that most of the requests concentrate on certain data blocks in a chunk. For example, overwritten pattern means high degree of local access of write requests. The access patterns are defined from one or multiple dimensions. To quantify the internal characteristics, we define four different classifications of chunk access patterns in Table 3.2. Taxonomy rule 1 is based on the intensity. Taxonomy rule 2 is based on the sequential and random degree. Taxonomy rule 3 combines two dimensions, randomness/sequentiality and read/write ratio. Taxonomy rule 4 differentiates various write patterns, considering sequentiality and local-access simultaneously.

Table 3.3: Summarization of I/O traces.

Trace	R/W (GB)	R/W IO Ratio	Access to Top 5% IO	Access to Top 20% IO	Access to Top 5% Data	Access to Top 20% Data
proj_2	1015.95/168.68	7.07	0.21	0.48	0.14	0.45
prxy_0	3.04/53.80	0.03	0.65	0.94	0.54	0.89
*hadoop*13	189.31/422.31	4.17	13.31	43.27	31.27	56.67
*backup*15	161.98/194.9	1.55	59.79	96.39	59.63	97.67

3.2.2 I/O Traces

We characterize a set of block level IO traces collected from two kinds of enterprise storage systems. The IO traces used in ChewAnalyzer include:

1. MSRC traces

The Microsoft Research Cambridge (MSRC) block-level IO traces are collected I/O requests on 36 volumes in 13 servers in about one week [71].

2. NetApp traces

Two kinds of NetApp block-level IO traces are collected on NetApp E-Series disk arrays [72]. One is from the Hadoop HDFS server running for about 8 hours. The other comes from the backup server running for about one day.

3.2.3 Observations, Analysis and Motivations

Observation 1: Access patterns in chunk level may be highly predictable over long durations.

Table 3.3 summarizes the I/O traces from the two storage systems. In most cases, most of the accesses concentrate on a small portion of the data. The IOPS of them are very high and we can use SCM or SSD to serve these chunks. Thus, we can improve data movements by mainly paying more attention to the active chunks. We set a tuple of thresholds to distinguish I/O intensities of taxonomy rule 1 in Table 3.1 (Non-access means zero access, while inactive means less than 2 accesses in a time window and the rest are active). Figure 3.1 shows the chunk access frequency cumulative distribution function for different I/O workloads. It shows the inactive data chunks, in most cases,

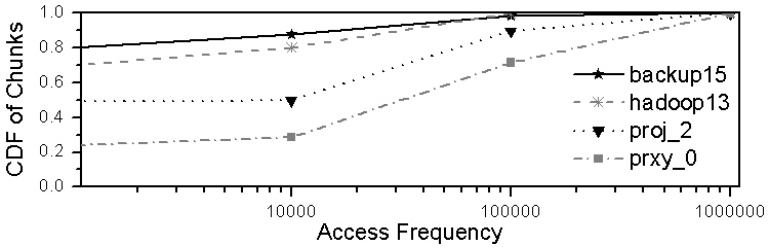
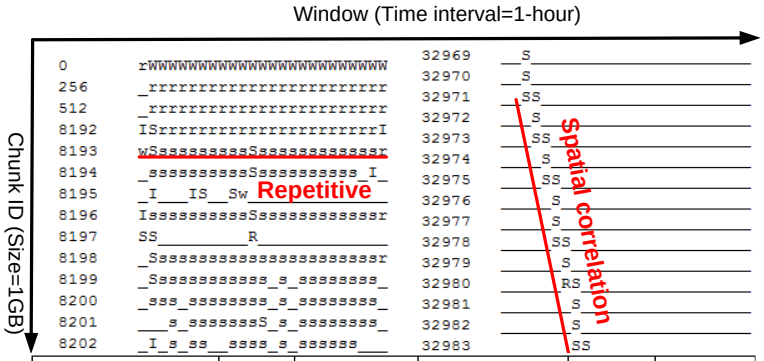


Figure 3.1: Chunk access frequency cumulative distribution function.

taking more than 90% of the total I/Os, are rarely accessed (less than twice in a 30-minute epoch). Therefore, we can place them on HDDs. More surprisingly, most of the data chunks have never been accessed in most of the periods. These data chunks, not only can be resided in HDDs, but also can be grouped together and managed with bigger chunk size to reduce metadata management overhead. For those few accessed chunks, we can group them together and maintain them in the slow speed but large capacity pools.

Previous work of workload analysis shows the active data set remains for long periods in most of the workloads [73]. This is similar to taxonomy rule 1 in Table 3.1. We study the access patterns of different workloads based on taxonomy rule 3. Sequential detection is implemented by identifying the distance of the Logical Block Address (LBA) of two consecutive requests (512 KB used in [17]). The results of analytical chunk-level characterization of these IO traces show that the access patterns of many chunks and some of the adjacent chunks are somehow inerratic in a period of time, which can potentially be utilized to optimize the data management in a heterogeneous storage system.

On the one hand, repetitive patterns happen in many chunks. For traces like *hadoop*13, *backup*15 and *proj_2*, the patterns are repetitive on a 10-minute, or 1-hour, or 1-day duration, we can somehow predict the future access pattern and do the corresponding data migration. The access patterns we analyze in this model tend to be stable over long durations. For example, the random write access pattern always concentrates on certain chunks and repeats from time to time. In *prxy_0*, we find 10% chunks are almost staying in a random write pattern. If the storage system remains enough free SCM space, thus it is the best that we can place them on SCM. On the other hand,



Pattern Name	Symbol	Ratio (%)	Pattern Name	Symbol	Ratio (%)
Non-access	'_'		Fully Rdm Read	'r'	7.71
Inactive	'i'	5.57	Fully Rdm Write	'w'	9.80
Seq. Read	'S'	15.23	Re-read	'R'	5.86
Seq. Write	's'	55.55	Overwritten	'W'	0.27

Figure 3.2: The diagram of chunk access pattern in consecutive time windows for *backup*15 trace.

the correlation between chunk accesses is similar as the spatial locality in cache study but in much longer periods. This observation inspires us to define proper patterns and match each of them to certain type of devices. Finally, we can use the pattern detection results to trigger new data placement.

Observation 2: Chunk access pattern may not change linearly.

The previous study of tiered storage always employs a heat-based approach [16] to guide the data movements between tiers. The most common metric is the access frequency (i.e., the heat) of a chunk [17]. Then data are moved based on the chunk temperature. Data access intensity is supposed to increase gradually. This makes the data movements in a safe and conservative way. However, the access intensity in certain chunks may not increase or decrease linearly due to bursty I/O. In some cases, the access pattern is only moderately hot and stable for a while without appearance of obvious intensive or cold patterns. Moreover, there are many cases that certain access patterns occur all of a sudden but disappear also quickly. Current big data processing applications have their own access characteristics. On a HDFS data server, IO requests are usually read sequentially from disks and accessed randomly over and over as shown in Figure 3.2. To analyze such cases, it is unsuitable to presume data access pattern will follow a hot to warm to cold manner.

Limitations of tier by tier data management: Data flows tier by tier may not be efficient when a storage system incorporates diverse storage pools. The overhead of moving data from one tier to the next tier involves not only data migration cost, but also the potential negative downstream/upstream data movement in the next tier. This may waste the I/O bandwidth between tiers. The data chunks in the downstream flow may lead to unnecessary data movement in a middle tier. If a chunk initially in the slow tier is becoming hot, it may be selected to move to the fast tier. If the fast tier is full, then one of the chunks in this tier may be affected. Applications or data which should run on this tier may be forced to migrate to the slow tier. If lots of chunks in one tier are selected to do migration, then the circumstance may be more exacerbating. Moreover, this tier by tier movement may result in higher latency if performed in long intervals.

Key insights: *From our experiments, we observe that there are many access patterns, such as sequential-write-only, write-once-random-read, moderate-intensive-random-read, etc. These patterns if detected to be predictive are proper candidates for directly mapping them to dedicated storage pools (one-to-one or one-to-several) without tier-by-tier data movements.*

Observation 3: Different dimensions and granularities affect profiling accuracy.

Using different dimensions in Table 3.1, we can get different workload profiling results. Following taxonomy rule 1 with different I/O intensities, the chunk accesses can be partitioned into certain groups. In a HDD + SSD system, we can use IO intensity to balance I/Os between these two pools [13]. According to Classification 2, we can partition data chunks into two groups, sequential and random patterns. When we study the architecture with HDD, SSD and SCM, some more parameters are to be considered. For SCM, overwritten (update) is not a critical issue. For SSD, random writes may increase internal write amplification and impact flash lifetime. While, for HDD, serving sequential write or write-once pattern improves system throughput and free the SCM and SSD space for intensive I/O patterns. In addition, different chunk sizes may generate obvious different results. Table 3.4 summarizes the ratios of different write patterns following taxonomy rule 4 for $prxy\_0$ and $hadoop13$ traces. We use two different chunk sizes, 20 MB and 100 MB.

Table 3.4: Summarization of different write access pattern ratios of *pxy_0* and *backup*15.

Chunk size	*prxy_0*			*backup*15		
	Seq. W	Fully Rdm W	Overwritten	Seq. W	Fully Rdm W	Overwritten
1GB	0.00	67.48%	13.34%	49.02%	10.36%	4.76%
128MB	0.00	49.66%	8.08%	52.30%	6.68%	4.03%

Key insights: Traditional approaches of I/O workload characterization for a two-tiered architecture are not sufficient for multiple storage pools. Accurate IO workload characterization is essential to fully utilize the capacity of each storage pool. The profiling accuracy is relative to which storage pools the workloads are running on. Thus, accurate storage-aware pattern exploration is useful to detect interior access patterns and eventually help the storage manager to place data into the best appropriate storage pools.

For some interesting chunks, we may need to further analyze and determine where they should go. In addition, as workloads move between the storage pools, the workload characteristics may be changed quickly. For examples, high frequently accessed small IO blocks may be better directed to flash-based SSDs. Infrequently but bustily accessed large IO blocks may be better kept on HDD or preloaded to high performance storage pools. Hence, the workload profiling should be aware of these changes and takes proper dimensions and granularities in each step of profiling.

3.3 The Framework of ChewAnalyzer

In this subchapter, we describe the ChewAnalyzer framework to conduct cost-effective data management across differentiated storage pools. To fully exploit the capability of each storage pool, the core part is devising an effective method of storage-aware chunk level workload profiling.

3.3.1 Overview

We consider the configuration with multiple storage pools, each of which is based on a different storage technology such as SMR, tape, traditional HDD, SLC-based SSD,

MLC-based SSD, SCM, etc. To overcome the shortcomings of tier by tier data management, they are fully connected so that data can move in between any two storage pools. Suppose we have n storage pools. Figure 3.3 describes the overview of ChewAnalyzer. The I/O Monitor keeps collecting the necessary I/O access statistical information in real time. It not only monitors the incoming I/O requests but also tracks the performance results of request executions on the storage devices such as the latency of each request and the queue length of each storage pool. Then, ChewAnalyzer analyzes the I/O access information through mainly two components, a Hierarchical Classifier (HC) and a Chunk Placement Recommender (CPR).

The Hierarchical Classifier [74, 75] is used to perform flexible access pattern classification for each chunk according to a set of taxonomy rules. ChewAnalyzer uses HC to find the proper chunks to place on the fastest pool (e.g. SCM) and then select chunks to place on the second-fastest pool until all the data chunks have been placed. A whole HC process is triggered during selecting data chunks on multiple storage pools (target pools). The HC groups data chunks into a set of categories on the basis of one taxonomy rule at each level. A taxonomy rule is defined by a certain dimension.

We build a benchmark tool to calculate the relative weight value in one dimension on various types of devices. Then we use a threshold to partition the chunks based on their weight value. After one classification level is finished, CPR will look up the status of the target pools and advises data placement policies based on the Pattern-to-Pool chunk placement library. If the available storage space and available performance capability in the targeted pools are sufficient for CPR to generate final data placement policies, chunk pattern classification and detection in HC will be stopped at this level. Otherwise, the next hierarchical classification in HC is triggered. After the HC ends, each chunk will be selected to place on a certain targeted pool. Finally, the placement policies are delivered to the Storage Manager that will make the final decisions of data movements. In the next we will describe these in details.

3.3.2 ChewAnalyzer

To classify a large set of chunks into different access patterns, ChewAnalyzer uses Hierarchical Classifier (HC) [74, 75] instead of flat classifier to group chunks level by level. To explain the HC methodology, we first introduce several notations shown in Table

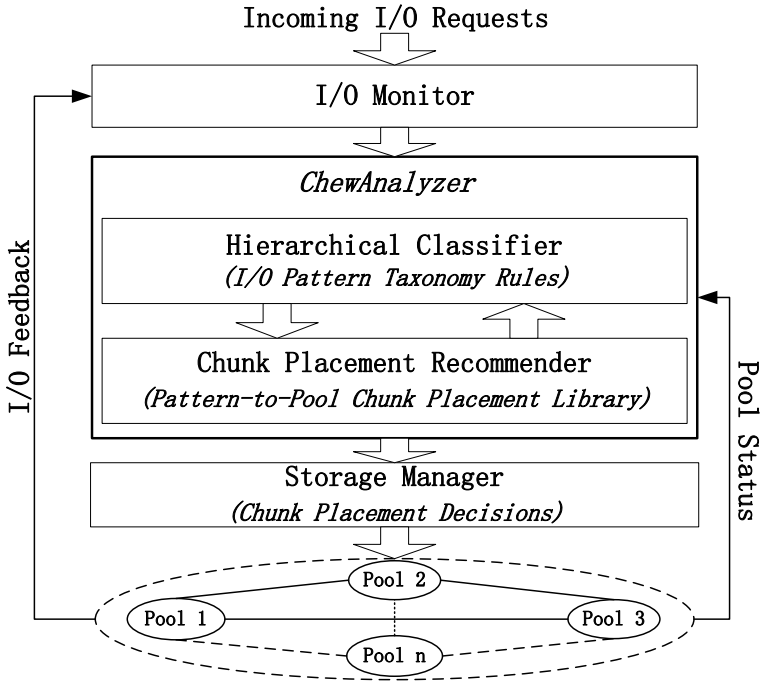


Figure 3.3: Overview of ChewAnalyer.

3.5.

We use the set, D, to represent all the used workload characterization dimensions in ChewAnalyzer, denoted as:

$$D = \{dimension_i\}, 1 \leq i \leq d \tag{3.1}$$

D includes the normal dimensions (e.g. intensity) in Table 3.1 for chunk level I/O access characterization. Then, at each level we define a taxonomy rule to partition the chunks into different groups on the basis of their access patterns. The dimension used in each taxonomy rule determines the results of partitions. The idiosyncrasies of each storage pool are respected on defining the taxonomy rules. To define a taxonomy rule, we introduce a metric, the relative weight in this paper.

Weight measurement: In [76], Michael Mesnier et.al. propose the relative fitness method of given a workload running on two different devices. To quantify the access differences among diverse storage devices, we develop a benchmark tool with preconfigured access patterns and run it on different devices to measure the performance. For

Table 3.5: Notations in ChewAnalzyer.

Notation	**Definition**
$chunk(i)$	The $i-th$ chunk
n	The total number of pools
x	The total number of chunks
$Size(i)$	The size of chunk i
d	The total number of dimensions for chunk characterization
n	The total number of storage pools
$Size(i)$	The size of chunk i
$FreeSpace(i)$	The total free space on the $i-th$ pool
$Performance(i)$	The current required performance value on the $i-th$ pool [17] (IOPS + throughput)
$Peak(i)$	The peak performance of the $i-th$ pool
$dimension\_i$	The $i-th$ dimension for workload profiling
h	The depth of HC
$PoolStatus(i)$	The status of the $i-th$ pool

each dimension, we calculate the relative weight for each type of devices. This process is called pattern-to-pool assertion, by treating each storage device as a black box. Then we compare the performance results to get the performance differences or namely weights. For example, if we have two storage pools, SSD and SCM read/write accesses perform differently on SSD. We run our benchmark tool to find the average read and write performance for SSD and SCM. Then we get the weights of a write request and a read request on SSD and SCM, respectively. Other factors, such as device lifetime, are also considered. For example, flash-based SSDs are more sensitive and fragile for write patterns. We amplify the weight value by $w0$. For example, the weight of write to read on SSD can be calculated as:

$$Weight\_ssd(write)' = Weight\_ssd(write) * w0 \tag{3.2}$$

The weight is calculated in a composited way. For example, we can get the weight of dimension read/write. If HC goes to the next level using dimension random/sequential, we use the weight combining both dimensions. When all the relative weight values are measured, we can calculate the total weight of the chunk in each level.

For example, in Level 1, for each $chunk(i)$, if we use $dimension(j)$ to characterize

the access pattern, the total weight value of this chunk can be calculated as:

$$Weight(i)(1) = The\,total\,weights\,based\,on\,dimension(j) \\ of\,all\,the\,accesses\,on\,chunk(i) \tag{3.3}$$

In Level k, for each $chunk(i)$, if we use $dimension(m)$ to characterize the access pattern, we need to combine the previous $k-1$ levels to calculate the relative weight value. Thus, the total weight value of this chunk can be calculated as:

$$Weight(i)(k) = The\,total\,weights\,based\,on\,all\,dimensions\,used \\ from\,Level\,1\,to\,Level\,k\,of\,all\,the\,accesses\,on\,chunk(i) \tag{3.4}$$

In the next, we will present pool status.

Pool Status: Chunk placement decisions are highly related to the pool status. If and only if the required resources of all the chunks have met in a pool, the placement decisions are valid. The performance ability of the pool is considered to achieve acceptable Quality of Service (QoS) for the requests. We employ the approximate performance metric [17], considering both IOPS and throughput. We define the current pool status as a AND operation of both free space and the available performance of this pool, denoted as:

$$Pool_{status(i)} = FreeSpace(i)\&\&(Peak(i) - Performance(i)), 1 \leq i \leq n \tag{3.5}$$

This includes the remaining free space of each pool as $FreeSpace(i)$ and the total available performance ability of each pool. If both are positive numbers, the placement policies are valid.

After we finish the computation of the weight value at a certain level, a set of thresholds based on the storage space of each pool are used to partition the chunks. These thresholds are usually set based on the Pattern-to-Pool chunk placement library.

Pattern-to-Pool Chunk Placement Library: We build a Pattern-to-Pool chunk placement library to guide ChewAnalyzer to make data placement decisions. This library takes into consideration of the idiosyncrasies of each storage pool. For example, an SSD pool may be suitable for random read accesses, but also proper for sequential read I/Os. For SLC-based flash pool, random write I/Os are still good candidates. There may be priorities for CPR to determine which one should be selected first. To

set each weight value, we can also consider the penalty of putting the chunks with such pattern into the pool, like the lifetime impact, power consumption and so on.

Figure 3.4 depicts the chunk pattern classification flow. When the chunk-level I/O access information arrives, ChewAnalzyer will choose one dimension to divide the chunks into several pattern groups. For example, if HC uses the I/O intensity dimension, it will partition the data chunks into n groups at most (as we have n pools) based on the number of I/O accesses. It may contain less then n groups as long as the storage pools can keep all the chunks. If the results are not sufficient for storage manager to make a data placement decision, another dimension will be added to partition the chunks in the next level. In Figure 3.4, if HC goes to the final $(h-th)$ level, ChewAnalzyer will select the chunks to different pools in the final level.

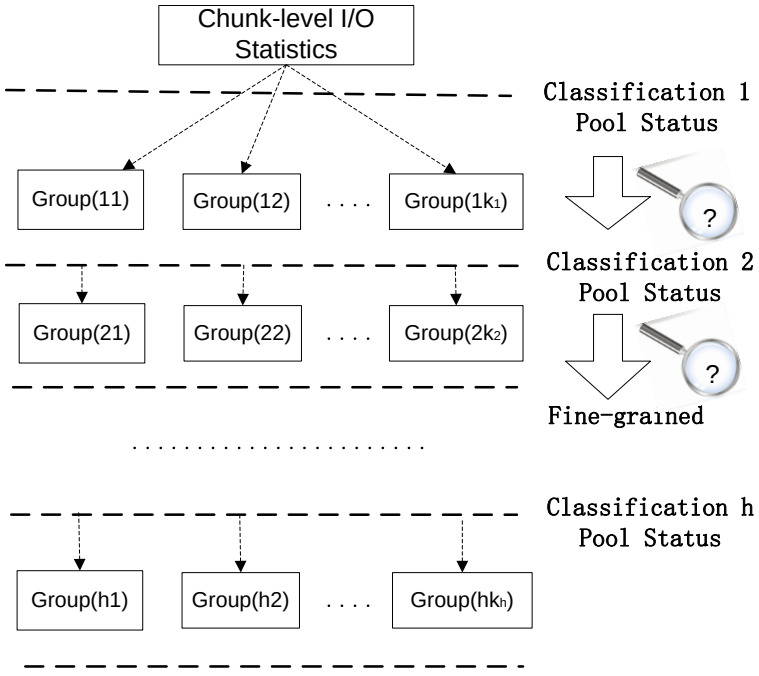


Figure 3.4: Chunk pattern classification diagram of Hierarchical Classifier.

The ordering of taxonomy rules to build the HC plays an important role on classification efficiency. ChewAnalzyer would like to fill with chunks on the fastest pool to fully make use of the resources. Thus, an ideal dimension list in HC first chooses one dimension to sieve the chunks which can be served the best on the fastest pool than

on the rest pools. If the sieved chunks cannot be kept on the fastest pool, HC will go to next level using another dimension. For example, for SCM pool, it is appealing to keep as many as data chunks which are going to be accessed, so we use I/O intensity to select the most active data chunks into SCM. Chunks with random write pattern are more proper to be placed on SCM than the ones which are random reads if both intensities are almost the same. In the next section, we will show a concrete example on how ChewAnalzyer implements HC.

ChewAnalyzer employs hierarchical taxonomy rules to classify and characterize chunks. The major advantage of using HC is to divide the chunks into different categories progressively. When HC has reached at the highest level, there may exist chunks that have not satisfied all the top placement policies from the Pattern-to-Pool library. Then, the placement decisions start from the fastest pool to the slowest pool to select a chunk based on the weight on that pool in the remaining list until that pool is full. Finally, each chunk will be placed on a certain pool. In most cases, without sufficient fast storage space, the rest ones will be placed on the large size storage pools. To make the ChewAnalyzer more flexible, the system designers may indicate more than one pool as the top priority placement decisions for certain patterns.

Chunk Placement: The placement decision of $chunk(i)$ in the $j-th$ pool is denoted as:

$$Placement(i) = j, 1 \leq i \leq x,\ 1 \leq j \leq n \tag{3.6}$$

where x is the total number of chunks and n is the total number of pools.

The goal of HC aims to exploit internal characteristics to assist the system identifying how well a chunk stays on each pool. As mentioned above, the detected patterns repeated in long durations can help the system to make decisions for data placement. The Chunk Placement Recommender (CPR) can perform cross-matching to place the chunk into the best appropriate pool. The cross-matching or namely storage-aware mapping depends on the properties of the pools.

3.3.3 Storage Manager

The placement policies are advised by CPR and then delivered to the Storage Manager. There may be urgent requests from the applications, while data movements may cause

the pool performance degradation for a while. In some worst cases, the migration requests may occupy the whole I/O communication and storage pools are not capable of completing the application requests. To respect the incoming I/O requests and the pool utilization, Storage Manager makes the final placement decisions. It periodically looks up the current status of all the storage pools. It also asks for the information of workload statistics from the I/O Monitor.

The way of scheduling migration depends on the current workload burden. To avoid I/O traffic of the foreground requests, the migration requests are first appended into a separate background process. Data migration process is wakened up when the target pool and the source pool both are available to allow performing data migration requests. A deadline is given to each migration request. If the deadline is expired before the next idle period, the new data migration policies will be re-considered. Moreover, we can use proactive migration [77] to first migrate a portion of the data for a big chunk. Then, I/O monitor can check the latency of the I/O requests in this chunk to confirm that the expected performance has been reached. Then, the reactive process is triggered to complete the rest migration.

3.3.4 Enhancements: ChewAnalyzer++

To make ChewAnalyzer more applicable and achieve more accurate profiling, we enhance ChewAnalyzer by dynamically changing the group granularity. If the chunk size becomes smaller, more interior information may be exploited. We call this enhanced process as ChewAnalyzer++. However, this may lead to too many smaller chunks in the system. To make this process more efficient and reduce the total complexity of chunk metadata management, we can select a subset of the chunks to do further characterization. The selected chunks are usually the candidates which are competitive for fast speed pool resources. For example, SCM is much expensive than SSD and HDD and we have to cautiously move new data into SCM, so it is valuable to use small chunk sizes in this type of storage pools. The splitting process could be recursive if the chunk placement recommender still has difficulty to make placement decisions. After the completion of partition, ChewAnalyzer++ can still follow the same procedure in HC and use CPR to advise new placement policies for the smaller chunks. We may define some fine-grained taxonomy rules to enhance the basic HC. Such fine-grained rules always employ more

than one dimension to represent the access patterns.

Although ChewAnalyzer++ can enhance the profiling accuracy by partitioning a big size into smaller chunks, it leads to more complicated data management. To avoid the size of most of the chunks becoming too small, we can merge the smaller size chunks into larger size chunks if they do not change their access patterns in a long period.

3.4 Case Study with A 3-Pool Architecture

In this subchapter, we will show the design and implementation of a 3-pool architecture as an example of employing ChewAnalyzer. We build a storage system composed of SCM, SSD and HDD pools. To make them fully connected, we first enable chunk level data movement between any of the two pools. The approximate performance numbers of the three pools are taken from Table 1. Their major features are summarized below:

- HDD pool: Concerning capacity, HDD provides large size storage pool with cheap price. A single HDD only supports 200∼500 IOPS for random requests, but sequential r/w throughput reaches 150MB/s.
- SSD pool: In SSD pool, the random performance is much better than that of HDD pool. IOPS for a single SSD arrives at ∼10k or more. However, random write requests bring extra garbage collection operations which not only decrease the performance but also influence the lifespan of flash memory. Thus, SSD pool prefers to serve random read pattern compared to random write requests.
- SCM pool: SCM pool provides the performance close to that of memory, but the price is the highest. SCM has good performance for requests with small size (less than 512 B) since it is byte addressable. We mount tmpfs on a fixed 4 GB DDR3 memory to build a ramdisk, which can achieve more than 1 million IOPS.

To calculate the peak performance *Peak(i)* of each pool, we use the combined IOPS and throughput model [17]. From the above analysis, the performance of HDD pool is mainly decided by throughput, while both SSD and SCM are much faster to support 10k ∼ 1million random IOPS. When sequential I/Os happen on the chunk which is placed on HDD, the remaining performance capacity is calculated based on the throughput. In the

contrary, if most of the I/Os are random on HDD, we use the IOPS metric to calculate the performance need. To identify the chunks which have the best suitability to stay in SCM pool, ChewAnalyzer deliberates the features of each pool and the workload pattern. In each level, ChewAnalyzer will use the pattern-to-pool mapping rules to make data placement decisions. In the meantime, we considers the total capacity and performance of the three pools.

To calculate the weight of various access patterns using a set of dimensions, we need to first design the HC. In the following, we will present the process of using HC to select chunks on the three pools.

To fully utilize system resources, ChewAnalzyer tries to use SCM to serve the incoming I/O requests as many as possible. This includes warming up SCM pool quickly by directly placing the chunks on SCM. Then if SCM is full, use SSD to place the proper chunks. To partition the chunks, we build a HC in Figure 3.5 using four taxonomy rules.

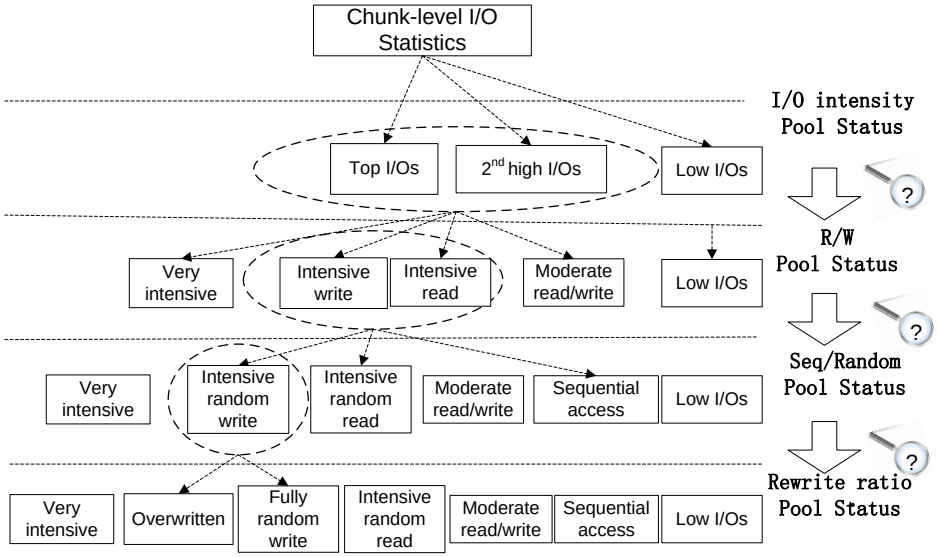


Figure 3.5: Diagram of using HC to classify data into different access pattern groups. (The weight of each chunk associated with each pattern is calculated in each level, so they can be ordered by their weight numbers)

As SCM has the best performance for both read/write I/Os in these three pools. We use I/O intensity as the first dimension to sieve the chunks which needs high performance pool. The requests are split into the same size of requests in our benchmark tool

(e.g. 512B). By descending the average I/Os per second (IOPS) on a chunk, we can group the chunks into these three groups, $Top\,I/Os$, $2nd\,high\,I/Os$ and $Low\,I/Os$. For example, we use the three IOPS ranges [300, +oo), [10, 300), [0, 10), to define $Top\,I/Os$, $2nd\,high\,I/Os$ and $Low\,I/Os$, respectively. If all the three pools can meet the Equation 3.5, the intensive data chunks are expected to be allocated on SCM, the second intensive data chunks are placed on SSD. The HDD pool is serving the chunks with few I/Os. If data placement decision cannot meet the free space constraint, HC goes to the next Level.

As SSD has better read performance than write performance. In addition, too many writes may cause SSD worn-out. We use the write/read dimension in Level 2 to further analyze the access pattern. The weight of each read operation is set to 1 unit. We amplify the weight of each write request by bringing a relative factor $Weight(w)$. If $Weight(w)$ is set to 2.0, the weight of each write operation is 2.0 unit. Then we can calculate all the weights of each chunk. Here, we do not calculate the $Low\,I/Os$ chunks for these chunks will be placed on HDD pool. We first select the chunks with fair large weights (weight in range [500, +oo)) to group into $Very\,Intensive$ pattern (e.g placing them on SCM). Then, we select the chunks with less weight value (e.g weight in range [300, 500), and partition them into two different patterns. If more than 50% of the accesses are write requests, they will be grouped into $Intensive\,write$ pattern. Otherwise, they will be grouped into $Intensive\,read$ pattern. The rest ones will be grouped into $Moderate\,read/write$ pattern (weight in range [10, 300)). We can place the $Very\,intensive$ ones into SCM. The $Moderate\,read/write$ and $Intensive\,read$ ones go to SSD. The $Intensive\,write$ ones will be suggested to stay on SCM. Then, we check the pool status to see if they can meet all the requirements. For example, if we have a very small SCM, only a part of the chunks in the $Very\,intensive$ group will be place on SCM. The rest ones will be place on SSD.

In Level 3, we use the randomness dimension to select the random access patterns from the $Intensive\,write$ and $Intensive\,read$ ones. The weight factor $weight(random)$ is used to amplify random access requests and filter out the $Sequential\,access$ chunks to HDD pool. Then we will check if SCM has enough space for the $Intensive\,random\,write$ ones. The $Intensive\,random\,read$ ones are placed on SSD.

In Level 4, ChewAnalzyer uses the rewrite ratio dimension to carefully allocates write requests on SCM and SSD. *Overwritten* pattern may produce more GC operations in SSD. Although, there may be asymmetric read and write problem for SCM, compared with SSD, SCM serves write requests much better. For the *Overwritten* pattern, a weight $w(overwrite)$ is used to amply their impact. If SCM does not have enough free space for the fully random write ones, they will be placed on SSD.

In previous tier by tier storage system or multi-level cache hierarchies, write-back strategy is always used to simplify this issue [17]. However, write back cannot solve the problem when write I/Os dominate in such a system. Write-back policies may incur synchronization overhead and consistency issue. To achieve peak performance, write requests may form a long queue and eventually increase the system latency. Moreover, write and read becomes more complex in a large scale storage environment when more and more applications are running. In some cases, data are written once but have few been accessed later, such as archival or back up applications. Some internet applications, updating may be very frequent, such as transaction processing, online-editing, etc.

We measure the weights of all these four dimensions using our benchmark tools. The benchmark tool incorporates all the typical I/O access patterns. We define the typical I/O access pattern in the benchmark program. The HC used in our design is based on a heuristic rule. Based on the HC design, we use the combined dimensions to get the relative weights.

Discussion of HC ordering: We can use different orderings to calculate the weights and do data placement. Each time the dimension used in the program makes a solution that is the best we can have. Though different HCs may generate different results, if HC stops in the mid-level, the results are valid data placement polices for this configuration. We still get the right data placemen. In the above description (Figure 3.5), we ignore the calculation of a portion of chunks. For example, ChewAnalzyer does not compute the write weight for the Low I/Os pattern in Level 2. Theoretically, we can calculate all the weight values of all the available dimensions. In the above design, we can still compute the Low I/Os chunks from Level 2 to Level 4. If HC stops at the last level, we will get the same results for different HCs.

We use the following pseudocode to describe the implementation of ChewAnalyzer and ChewAnalyzer ++.

Algorithm 1 ChewAnalyzer

Input: $chunk(i)$, $1 \leq i \leq x$
Output: $\{Placement(i), 1 \leq i \leq x\}$

1: $k \leftarrow 0$
2: **while** $k <= h$ **do** //h is the maximum level in HC
3: **for** Each $chunk(i)$ in the current HC **do**
4: Add $chunk(i)$ to a pattern group $\leftarrow$ *Pattern detection algorithm*
5: **end for**
6: **for** Each $chunk(i)$ **do** //placement policy based on the top priority decision
7: $j \leftarrow$ the first priority pool based on the Pattern-to-Pool Chunk Placement Library
8: **if** Equation 3.5 is TRUE **then**
9: $Placement(i) \leftarrow j$ //Place $chunk(i)$ on $pool(j)$
10: **else**
11: $k \leftarrow\ k + 1$
12: **goto** nextLevel
13: **end if**
14: **end for**
15: **if** $k \leq h$ **then**
16: **goto** RETRUN
17: **end if**
18: **nextLevel:**
19: **end while**
20: **do** greedy sorting all the rest of the chunks which have not been placed based on their weights
21: **RETURN:**
22: **return** $\{Placement(i), i\epsilon\{1, n\}\}$

In Algorithm ChewAnalyzer, the input data is a set of chunks in a time based sliding window, which keeps moving after an epoch (e.g. 10-minute) in the run-time. At the end, ChewAnalyzer outputs the placement decisions of all the chunks. The core of HC is implemented from Step 1 to Step 19. Based on the I/O pattern taxonomy rule in each level, the chunk detection model is used to decide the chunk pattern value (step 4). If chunk(i) is not affected, it will remain in the previous pattern group. In Step 7, the placement decision is based on the pattern-to-pool library. If Equation 3.5 returns TRUE (Step 8), then chunk(i) is placed in pool j (Step 9). If all the chunks with the pattern in a certain level in HC can be placed based on the pattern-to-pool library, HC will be ended up (Step 16) and directly goes to Step 21. After performing the chunk

Algorithm 2 ChewAnalyzer++

Input: A set of chunks with initial pattern value of $chunk(i)$ from Algorithm 3.4 (ChewAnalyzer) $i\epsilon\ \{1, x\}$

Output: $\{Placement(i),\ i\epsilon\{1, y\}\}$ //chunk placement policies $(y >= x)$

1: $SubChunkSet \leftarrow \emptyset$
2: **for** Each $chunk(i)$ in the current chunk set **do**
3: **if** $pool(j)$ is SSD or SCM **then**
4: **for** every $chunk(i)$ in $pool(j)$ **do**
5: **if** $chunksize(i) > Ch0$ **then**
6: Partition chunk(i) into p sub chunks evenly //p-partition
7: $SubChunkSet \leftarrow$ *add new sub chunks* //remove the original $chunk(i)$
8: Update $Pool_{status(j)}$
9: **end if**
10: **end for**
11: **end if**
12: **end for**
13: **Call:** Algorithm3.4($SubChunkSet$)
14: **return** $\{Placement(i), i\epsilon\{1, y\}\}$

pattern classification in HC and pattern-to-pool mapping, each chunk will be given a certain access pattern, which may be composited with different dimensions. If a portion of chunks cannot be placed based on the first priority at the end of HC, in Step 20, a greedy sorting algorithm is used to select the chunks with the most matched patterns to reside in each pool. If there is not enough space in the fastest pool, the following chunks will be placed on the less high performance pool. Eventually, all the chunks will be placed in a certain pool. The placement decisions are returned in Step 22 at the beginning of each sliding window.

In Algorithm ChewAnalyzer++, it first selects the chunks whose size is larger than $Ch0$ in SCM and SSD and add them into the SubChunkSet (Steps 1-5). Then, these chunks are partitioned into p sub chunks (Step 6). Here, we can change the partition parameter p to divide big size chunks to different numbers of sub chunks each time. This can be improved by treating SSD and SCM pool using different granularities in ChewAnalyzer++. We can also do recursive partition to reach the optimal solution. When the partition process is finished, the original chunk is replaced with p new sub chunks in SubChunkSet (step 7). In addition, the freespace and performance of each

pool are updated (step 8). In step 13, ChewAnalyzer++ will call ChewAnalyzer by inputting all the chunks in the SubChunkSet. In step 14, the new data placement polices based on the selected sub chunks are returned.

3.5 Evaluation

We evaluate ChewAnalyzer by means of replaying real enterprise block I/O traces on a Linux platform as shown in Figure 3.6. We build a three-pool prototype based on Linux device mapper (DM) [37]. The Linux kernel version is 3.13.0. The heterogeneous storage controller works as a standard Linux block driver in DM which manages the block devices in a linear manner. Our trace replay engine is implemented via libaio[78] in user level in Linux. We enhance the heterogeneous storage controller to perform chunk level data migration using the "kcopyd" mechanism in DM [38]. Chunk migration manager communicates with ChewAnalyzer though sysfs [46]. Data chunks are triggered to be migrated from one pool to another. To maintain the mapping after migration, we record the chunks using a mapping table in memory. Each tuple contains the logical chunk ID and physical address chunk ID on a pool. The Dell Power Server used in our experiment is configured with a Seagate 8 TB HDD drive, a Samsung 850 pro 512 GB SSD, 48 GB DDR3 memory and an Intel Xeon E5-2407 2.20 Ghz CPU.

Sequential detection is non-trivial. In our evaluation, sequential I/O patterns have been carefully classified and detected [69]. The baseline methods we use to compare with ChewAnalyer and ChewAnalyzer ++ contain the following two existing policies:

- Greedy IOPS-only Dynamic Tiering (IOPS-only): IOPS-only is used widely as a heuristic approach to group hot data. This policy measures the IOPS on each chunk to migrate data chunks tier by tier.
- EDT [17]: EDT migrates fixed size data chunks tier by tier based on the combined weight from throughput and IOPS on each chunk.

We run the four policies with four different data traces and compare the results in the next. Table 3.2 summarizes the four traces, $prxy\_0$, $proj\_2$, $hadoop13$ and $backup15$. The first two are from a data center in Microsoft Cambridge in 2008. The other two traces are from the E-series array of NetApp [72] in 2015. One was running with a

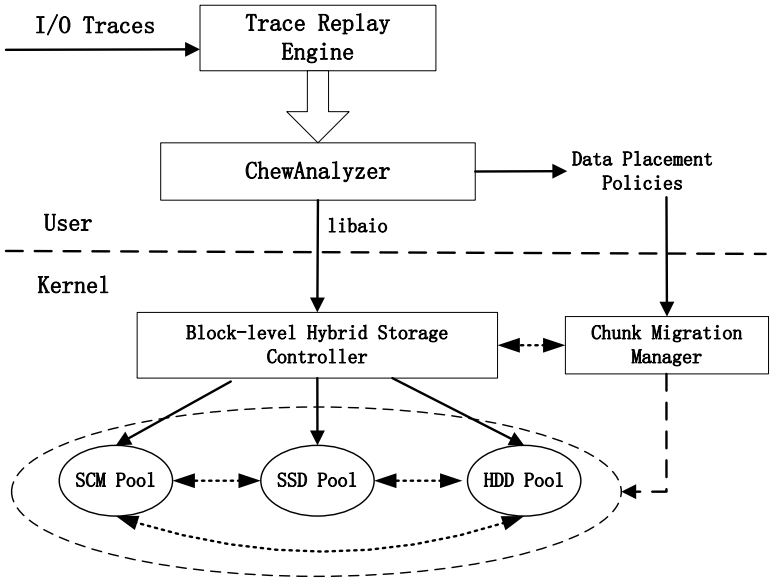


Figure 3.6: Trace-driven storage prototype for evaluating ChewAanlyzer.

Hadoop server and the other was running with a backup server. To carry out migration, time is divided into fixed-size epochs (sliding windows; 45-minute for *prxy_0*, 1-hour for *proj_2*, 20-minute for *hadoop*13 and *backup*15). In addition, the total number of requests in window is set to less than 20k. Chunk size is set to 256 MB. The minimal size of a chunk in ChewAnalzyer++ is set to 64 MB.

ChewAnalzyer only uses fewer dimensions to classify data chunks. This reduces the metadata management cost for storage. In the following experiments, the storage configurations as (SCM, SSD) for *hadoop*13, *backup*15, *proj_2* and *prxy_0* are (10 GB, 100 GB), (10 GB, 100 GB), (10 GB, 40 GB) and (2 GB, 10 GB) respectively.

Let us look at the performance results after employing the four different policies. Figure 3.7 shows the normalized average I/O latency when we replay the four traces. In these four cases, IOPS-only have longer latency than the other three policies. Both ChewAnalyzer and ChewAnalyzer++ outperform EDT and IOPS-only by shorter I/O latency. Compared with IOPS-only, ChewAnalzyer accelerates I/O access by 26%, 13%, 24% and 23% respectively for 4 different configurations. ChewAnalzyer++ further reduces I/O waiting time by 4.5%, 5.4%, 8.4% and 16.7% respectively. The major reason is that both IOPS-only and EDT do not perform fine-grained access pattern

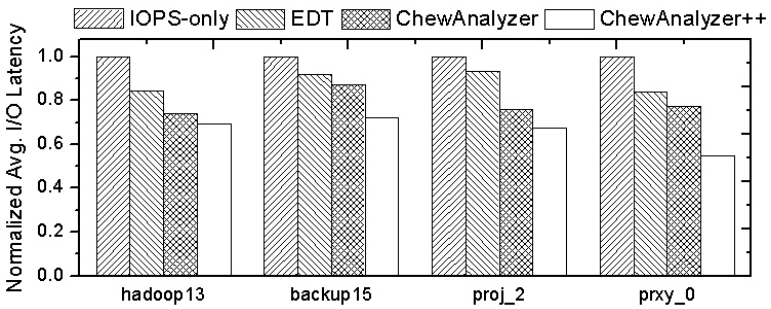


Figure 3.7: Normalized average I/O latency for the four policies.

classification. With HC, ChewAnalyzer (++) exploits more access pattern dimensions to assist data placement across different storage devices. ChewAnalzyer++ selectively partitions chunks to profile their internal patterns so as to make better utilization of SCM and SSD.

As the prototype emulates SCM by DRAM, which is regarded as the costliest part. Decoupling small overwritten I/Os and placing them on SCM not only increases the overall performance but also reduce the write load on SSD. ChewAnalyzer++ further improve profiling accuracy by looking into a portion of chunks which may require high performance storage resources. ChewAnalzyer (++) makes decisions at the beginning of each sliding window. This helps the storage manager filter out the data which does not need to run on high performance pool. Let us look the pattern ratio distribution in ChewAnalyzer and ChewAnalzyer++. Figure 3.8,3.9,3.10,3.11 show the accumulated overwritten pattern ratio change in each sliding window when replaying the four traces,respectively. The x axis represents the t-th sliding window. For the backup trace, accesses are mainly read or sequential write, so both have almost the them same ratios of overwritten pattern chunks. In hadoop trace, both recognize a small portion of data chunks as overwritten, ChewAnalzyer++ still exploits deeper than ChewAnalzyer. In some cases, there is no overwritten pattern. Part of the reason is that HC returns before Level 4. For *proj_2* trace, overwritten ratio greater than 5% happens only in 3 sliding windows. As random writes dominate in *prxy_0*, ChewAnalzyer++ reduces overwritten ratio in each sliding window from 58% to 45%.

In all the four policies, data migration is triggered at the start of each sliding window. Moving data from one pool to another pool leads to extra read and write cost for the

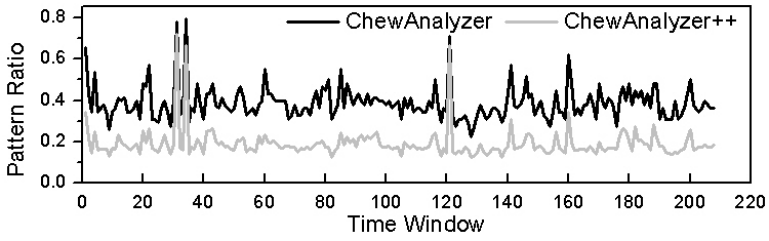


Figure 3.8: Overwritten pattern ratio at different time window ($prxy\_0$).

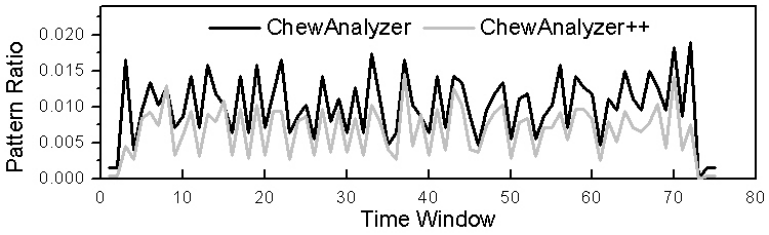


Figure 3.9: Overwritten pattern ratio at different time window ($hadoop13$).

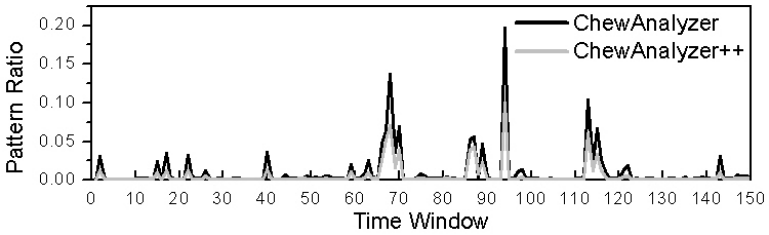


Figure 3.10: Overwritten pattern ratio at different time window ($proj\_2$).

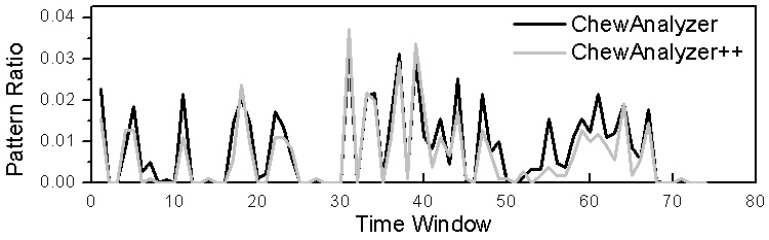


Figure 3.11: Overwritten pattern ratio at different time window ($backup15$).

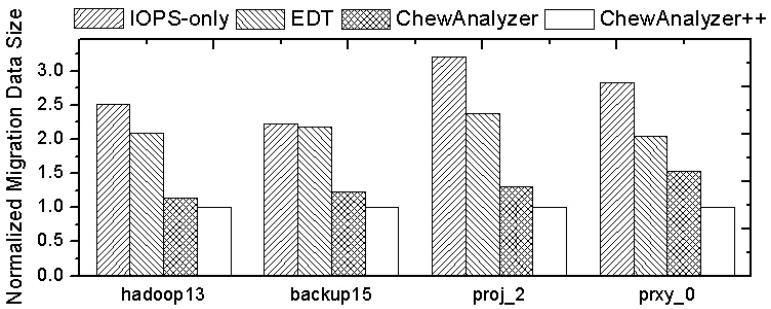


Figure 3.12: Normalized total data migration size.

storage system. In addition, the migration overhead hurts overall performance if storage devices are connected via external network (e.g. remote cloud). Figure 3.12 depicts the normalized total amount of data being migrated. Both IOPS-only and EDT performs data movement tier by tier which may incur extra migration load on the moving path. With fine-grained access pattern classification, both ChewAnalzyer and ChewAnalzyer++ can directly move data chunks and thus alleviate the migration load. For the backup trace, both IOPS-only and EDT have almost the same amount of migration load. While, ChewAnalzyer and ChewAnalyzer++ have 45% and 55% less migration load compared with IOPS-only respectively. ChewAnayzer and ChewAnalyzer++ reduce the total amount of data migrated by 46% and 64% compared with IOPS-only in $prxy\_0$ trace respectively.

Flash based SSD drives have asymmetric read and write speeds. Moreover, flash has the wear-out problem which affects the lifetime of the drives. In the configuration of the Chunk-to-Pattern data placement library, ChewAnalzyer prioritizes the write-dominated chunks on SCM to reduce the write load on SSD. Finally, let us calculate the write requests distribution on the three pools. Figure 3.13 shows the normalized write request distribution on each pool for the $prxy\_0$ trace. ChewAnalzyer (++) increases the write load in SCM and reduces the total write times on SSD. ChewAnalyzer++ aggregates almost 85% of write I/Os on SCM and makes the least number of write requests on SSD ($\sim 10\%$). This significantly reduces the write load on SSD and improves its lifetime.

In summary, our experiments show that both ChewAnalzyer and ChewAnalzyer++

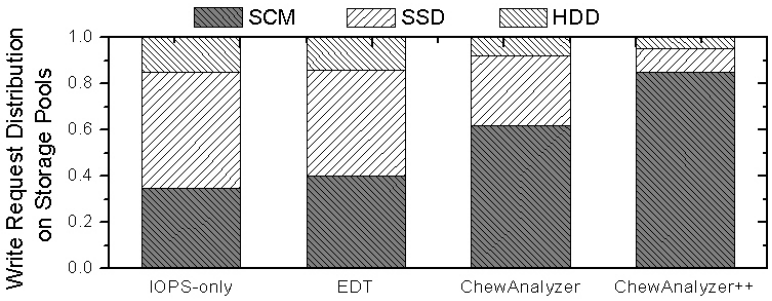


Figure 3.13: The average latency on a hybrid SSD and HDD volume.

outperform the other two policies by shorter average I/O latency. In addition, they reduce the migration load and the random write load on SSD. ChewAnalzyer++ partitions the data chunks on SCM or SSD to further assist efficient data placement decisions.

3.6 Related work

In the following, we summarize the related work and compare them with ChewAnalyzer.

3.6.1 Tiered Storage Management

Recent research has focused on improving storage cost and utilization efficiency in different storage tiers. Guerra et al.[17] build a dynamic tiering system that combines SSDs with SAS and SATA disks to minimize cost and power consumption. IOPS and throughput are considered to carefully control the overhead due to extent migration. CAST [16] provides insights into the design of a tiered storage management framework for cloud-based data analytical workloads. File level based HSM [62, 63] takes advantage of the fact that data are not the same value during any given period of time for applications. In HSM, data migration conditions are typically based on ages of files, types of files, popularity of files and the availability of free space on storage device [62]. However, the mechanism used in prior approaches cannot be directly applied to the fully connected differentiated storage pools that including SCM, SSD and HDD, mainly becuase they generally ignore the special read and write properties between SSD and SCM.

3.6.2 IO Workload Characterization

The theory of IO workload characterization provides a useful abstraction for describing workloads more concisely, particularly with respect to how they will behave in hierarchical storage systems. A large body of work from the storage community explores methods for representing workloads concisely. Chen et al. [79, 80] exploit workload features via machine learning techniques. Tarasov et al. extract and model large block I/O workloads with feature matrices [81]. Delimitrou et al. model network traffic workloads using Markov Chains [82]. Bonfire [83] accelerates the cache warm up by using more efficient preload methods. Windows SuperFetch [84] preloads the frequently used system and application information and libraries into memory based on the usage pattern in history to reduce the system boot and application launching time. Cast [16], a storage tiering framework performs cloud storage allocation and data placement for analytical workloads to achieve high performance in a cost-effective manner. While these related work focuses mostly on identifying the data that should be brought into the fast tier for higher efficiency only based on IOPS and intensity of workloads, ChewAnalyzer concentrates on seeking a chunk level data management to efficiently incorporate differentiated storage pools, including SCM, flash-based SSD, and HDD, which can make enterprise storage hierarchies more efficient with diverse storage pools.

3.6.3 SSD and SCM Deployment in Storage Systems

NAND flash memory based solid state drives (SSD) see an increasing deployment in storage systems over the last decade, due to its advantages, such as light weight, high performance and low power consumption. However, the limited P/E cycles may accelerate wear-out of flash chips in SSDs, which is always a potential reliability issue in SSD based storage systems [18, 19]. The relative high cost of write operations is still the performance bottleneck of flash memory. Hybrid HDD and SSD storage sytems have been extensively studied [85]. GREM presents dynamic SSD resource allocation in a virtural machine envrioments [86]. Today's PCI-e based SCMs represent an astounding 3X performance increase compared with traditional spinning disks ($\sim$100K IOPS versus $\sim$100) [6]. To maximize the value derived from high-cost SCMs, storage

systems must consistently be able to saturate these devices. Despite the attractive performance of these devices, it is very challenging to effectively slot them into existing systems. Hardware and software need to be designed together with an aim of maximizing efficiency. In ChewAnalyzer, we better utilize the unique price-performance tradeoffs and idiosyncrasies of SCM, SSD, and HDD in a fully connected differentiated storage system. ChewAnalyzer also allocates/migrates data with respect to workload characteristics each storage device prefer to support.

3.7 Conclusion

In this chapter, we study the architecture with differentiated storage pools fully connected to suit diverse workload profiles. To explore the internal access patterns and thus efficiently place data in such a complete topology, we propose a Chunk-level storage-aware workload analyzer framework, simplified as ChewAnalyzer. Access pattern is characterized as the collective accesses in a chunk composed of a set of consecutive data blocks. The taxonomy rules are defined in a flexible manner to assist detecting the chunk patterns. In particular, ChewAnalyzer employs Hierarchical Classifier to exploit the chunk patterns step by step. In each classification step, the chunk placement recommender advises new data placement policies according to the device properties. Both pool status and device properties are considered on making placement decisions. ChewAnalyzer++ is designed to enhance the workload profiling accuracy by partitioning selective chunks and zooming in their interior characteristics. According to the analysis of access pattern changes, the storage manager can adequately distribute the data chunks across different storage pools. ChewAnalyzer improves initial data placement and if needed migrates data into the proper pools directly and efficiently. We build our prototype equipped with Storage Class Memory (SCM), Solid State Drive (SSD) and Hard Disk Drive (HDD) in a Linux platform. Through trace driven approach, our experimental results show ChewAnalyzer outperforms the conventional dynamical tiering by less latency and less write times on the flash pool. The total amount of data being migrated is also reduced. In this study, we did not consider the connection costs and the available bandwidths between different storage pools. These considerations will be included in our future studies.

Chapter 4

Flexible and Efficient Acceleration for Network Redundancy Elimination

To reduce the duplicate content transferred between local storage devices and devices in remote data centers, Network Redundancy Elimination (NRE) aims to improve network performance by identifying and removing repeated transmission of duplicate content from remote servers. Using a Content-Defined Chunking (CDC) policy, an inline NRE process can obtain a higher Redundancy Elimination (RE) ratio but may suffer from a considerably higher computational requirement than fixed-size chunking. Additionally, the existing work on NRE is either based on IP packet level redundancy elimination or rigidly adopting a CDC policy with a static empirically-decided expected chunk size. These approaches make it difficult for conventional NRE MiddleBoxes (MB) to achieve both high network throughput to match the increasing line speeds and a high RE ratio at the same time. In this chapter, we present a design and implementation of an inline NRE appliance which incorporates an improved FPGA-based scheme to speed up CDC processing to match the ever increasing network line speeds while simultaneously obtaining a high RE ratio.

4.1 Introduction

More and more emerging Internet applications are driving a boom of Wide Area Network (WAN) bandwidth requirement. Wide Area Network (WAN) Optimization Accelerators (WOAs) have been widely developed in the form of MiddleBoxes (MBs) by multiple vendors to optimize network transmission efficiency between data centers and clients [87, 88]. A considerable amount of network traffic is usually repeatedly transferred across networks when different users on the Internet access the same or similar content [20, 21]. To reduce the overall network bandwidth requirements, Network Redundancy Elimination (NRE) plays a primary role in a WOA by identifying and removing repetitive strings of bytes across WAN connections. Glancing ahead to Figure 4.11 will help the reader visualize how NRE MBs fit into a WAN. Generally, a higher Redundancy Elimination (RE) ratio can save more bandwidth [23].

For an inline NRE process, the content of network flows is first segmented into a set of data chunks. Then these data chunks will be identified to be redundant (i.e., has been recently transmitted and buffered) or not. Chunking policies based on either fixed or variable sizes determine the RE ratio [20]. Compared with a fixed-size chunking policy, a variable-size chunking policy can more efficiently identify repetitive chunks. Content Defined Chunking (CDC) [24], a variable chunking policy, has been widely used by many NRE solutions [21, 25, 26]. However, some components of CDC consume significant CPU time (e.g., the Rabin hash process). This overhead will affect the server performance and eventually decrease the overall NRE throughput. For instance, considering a standard software-based NRE MB (Intel E5645 CPU, 2.4 GHz, 6 cores, exclusive mode), the CDC chunking throughput is about 267 Mbps for each core and totals around 1.6 *Gbps* [27]. Now consider two typical examples of NRE-based WOA products: the Cisco WAE-7371 [89] and the Infineta DMS [90]. While both these accelerators are faster than the software-based MBs, the WAE-7371's throughput is about 2.5 *Gbps* and lags far behind that of the DMS, which is about 10 *Gbps*. This is because the DMS adopts a fixed-size chunking policy to guarantee high throughput at the expense of lowering the RE ratio while the WAE-7371 uses CDC for a higher RE ratio. For a CDC scheme, there is also a tradeoff between RE ratio and the expected data chunk size. The smaller the expected chunk size, the higher the RE ratio. However, the smaller expected chunk size

will require higher computational cost.

Some effort has been made to balance the tradeoff between throughput and RE ratio in NRE [20, 91], but improvement is still limited when attempting to achieve both using software alone. Separately, one trend in networking research is the use of field-programmable gate arrays (FPGAs) to improve the performance of specified network functions [92–94]. However, combining these ideas to make an FPGA-based NRE appliance still needed to be explored. In addition, previous work on NRE design usually employs an immutable CDC policy with an empirically-decided expected chunk size [21, 23, 25, 95]. For some TCP flows, however, the CPU-bound CDC scheme is not necessary when the fixed-size chunking policy performs only a little worse than CDC in RE ratio but provides much better network throughput. One of our goals is to design an NRE that can dynamically decide between CDC or fixed-size chunking policies, and also accelerate CDC when in use, therefore improving RE ratio and throughput for varying workloads.

This chapter presents a specially-designed NRE appliance with an FPGA substrate. We design a Virtualized Network Redundancy Elimination (VNRE) controller to efficiently and flexibly utilize the hardware resources. During the computation process, VNRE uses an improved FPGA-based scheme to accelerate the CDC process. The Rabin fingerprint (FP) computation [24] is offloaded through a record table and the FPGA registers. To our knowledge, this work is the first to implement Rabin FP and CDC in an FPGA in a reconfigurable fashion with dynamically-adopted chunking policies. That is, our novel VNRE monitors each TCP flow and customizes the following two NRE parameters:

- **Chunking policy**. The chunking policy can be either fixed-size or CDC. Various network flows show differing degrees of redundancy [20, 91]. The fixed-size chunking policy performs well for multimedia file transmission (e.g., disk images [96], video and audio [97], etc.), while the CDC chunking policy is preferable for HTTP and other text-based file transmission [20, 26].

- **Expected chunk size**. This tuning parameter determines both network throughput and RE ratio. Some applications (e.g., HTTP) are sensitive to chunk size, while others (e.g., RTSP and encrypted HTTPS) get little RE ratio improvement

when using smaller chunk sizes [91]. In general, however, the smaller the expected chunk size, the higher the RE ratio and the lower the network throughput.

We complete a prototype of the VNRE controller in a standard x86 server with a Partially Reconfigurable (PR) FPGA card [27, 98]. Through trace-driven evaluations, our experimental results can be concluded as follows:

- Compared with the baseline configuration (1 virtual CPU (vCPU), 16 GB RAM, and a 1 TB disk as a software based NRE MB), the CDC scheme performed by a PR unit improves the network throughput by nearly 3X.
- For each TCP flow under the VNRE-controlled configuration, the overall throughput outperforms multiple static configurations by 6X to 57X.

The rest of this chapter is organized as follows. Chapter 4.2 provides the background and motivations. Chapter 4.3 describes our VNRE design. In Chapter 4.4, we present the performance evaluation method and discuss the experimental results. Chapter 4.6 summarizes the related work. Chapter 4.5, we present OpenANFV to accelerate Network Function Virtualization (NFV) with a consolidated framework in OpenStack. Finally, we conclude this work in Chapter 4.7.

4.2 Background and Motivations

As shown in Figure 4.1, a typical NRE process includes three basic stages. In Stage 1, a network flow is split into a sequence of data chunks through a customized chunking policy. Within a chunking policy, a critical factor is a pre-determined expected chunk size. In Stage 2, an FP is usually generated by a cryptographic content-based hash algorithm, such as SHA-1 or MurmurHash [99], consuming much CPU time. In Stage 3, NRE can determine whether or not the current data chunk exists and is being stored locally based on the index. Stage 1 and Stage 2 are usually processed on the server side while Stage 3 is processed on the client side.

Stage 1: Chunking. A network flow is split into a sequence of data chunks through a customized chunking policy (fixed-size or variable-size). Within a chunking policy, an important factor is a pre-determined expected chunk size. Technically, we can adopt

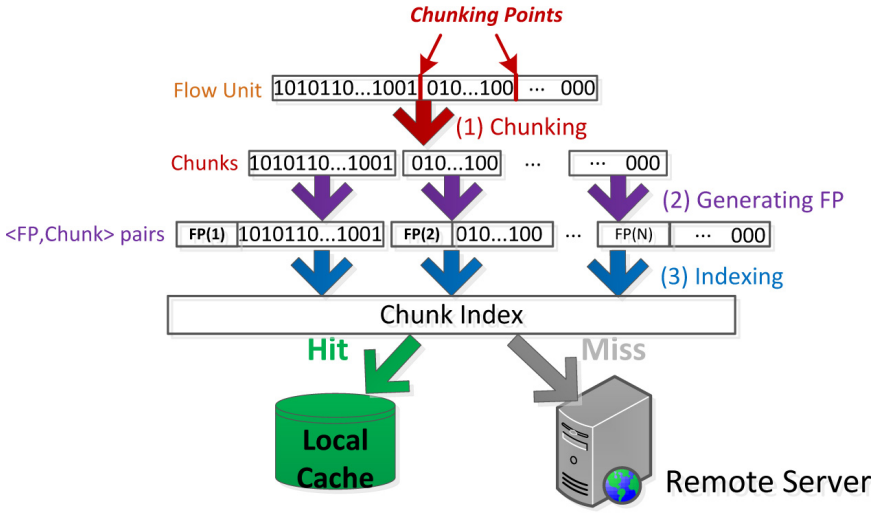


Figure 4.1: Simplified flow diagram of a NRE process.

fixed-size chunking and variable-size content-based chunking to define the chunk boundary. Usually, the granularity of a flow unit is either on the IP packet level or TCP flow level.

Stage 2: Generating Fingerprint (FP). The FP of a chunk is regarded as the key for comparison and identification. A FP is usually generated by a cryptographic content-based hash algorithm, such as SHA-1, MD5, or MurmurHash [99]. A FP needs to be large enough to uniquely identify the represented data chunk, and the process of generating FP is time consuming. The probability of hash collision in those widely-used hashing algorithms can be neglected. Although content-based hashing algorithms are CPU-bound, the computation cost of FPs and using them for chunk comparisons is still much faster than comparing chunks bit by bit.

Stage 3: Indexing and Storing Data Chunks. The purpose of indexing is to quickly determine whether or not the current data chunk exists and is being stored. If a data chunk already exists, it does not need to be delivered again. If a data chunk is unique (does not currently exist), it can be stored and indexed. Of course, the number of data chunks that can be indexed and stored depends on the available storage space. Additionally, an inline process for fast indexing and retrieval always incurs intensive I/Os.

4.2.1 FPGA Accelerator and Quick Prototyping

A field-programmable gate array (FPGA) is comprised of an array of programmable logic blocks with a series of reconfigurable interconnects. Leveraging the hardware parallelism of FPGA devices, we present an FPGA-based acceleration scheme that is applied to speed up the CDC algorithm in the NRE process. FPGA technology can provide the flexibility of on-site programming and re-programming without going through re-fabrication of a modified design. One of our design goals is to fully exploit this capability by building our prototype on a Partially Reconfigurable FPGA (PR-FPGA) to quickly verify and evaluate multiple alternative acceleration schemes and adopt the one that is the most beneficial. A PR-FPGA contains several modular and isolated processing units, which we will call PRs, that allow heterogeneous functions to be switched and reloaded quickly. This transition can occur in the same FPGA by loading a partial configuration file, usually a partial bit file, without affecting other PRs and the static regions. Reload time of a partial reconfiguration is much smaller (3-5 ms) than full reconfiguration (10-15 ms). Therefore, the application of this PR-FPGA technology in our prototype further enhances system flexibility. Since this chapter primarily concentrates on how to selectively and efficiently accelerate the NRE process by using an FPGA substrate, we do not discuss the details of the resource management for different PRs and the further components of PR reconfiguration cost.

4.2.2 The Role of Chunking Policy

The chunking policy is used to identify chunk boundaries. Fixed-size chunking is very sensitive to content updates. If only one byte is inserted into the data flow, all the remaining fixed-size chunks behind the updated region look different. CDC segments a flow into variable-size chunks through the canonical Rabin hash algorithm [100]. In the Rabin hash generation process, a small and fixed-size sliding window (usually 12$\sim$64 bytes) is used to calculate the hash value F. Let us denote the window size as M bytes and the numerical ASCII value of each byte in the first M-bytes in sequence as $t_1, t_2, ..., t_M$. The initial hash value F_1 of the first sliding window is calculated as Equation 4.1, where R indicates the digital base value (e.g., $R = 10$ if the ASCII is represented as decimal or 16 for hexadecimal as is our case). Q is set as a large prime

Table 4.1: Throughput and RE ratio of an *NFSv4* server using different chunking policies and average chunk size.

Chunking policy	**Expected chunk Size**	**Processing Unit**	**Throughput**	**RE Ratio (%)**
Rabin hashing based	MODP (32 B)	IP packet	52.0 Mbps	56.2%
	MAXP (32 B)	IP packet	46.4 Mbps	59.7%
	CDC (128 B)	TCP flow	113.6 Mbps	48.8%
	CDC (512 B)	TCP flow	507.2 Mbps	45.7%
	CDC (2 KB)	TCP flow	2.46 Gbps	32.4%
	CDC (8 KB)	TCP flow	7.63 Gbps	24.8%
Fixed-size based	128 B	TCP flow	246.4 Mbps	29.2%
	512 B	TCP flow	979.2 Mbps	24.6%
	2 KB	TCP flow	4.81 Gbps	19.7%
	8 KB	TCP flow	28.0 Gbps	17.5%

number. The hash value of the $(i+1)$th window, F_{i+1}, is calculated via Equation 4.2 according to the previously calculated value F_i and Horner's rule [24]. A record table can be established to record all 256 possible values of $(t_i \cdot R^{M-1} \ mod \ Q)$ and allows the use of a lookup operation to speed up the whole calculation. Supposing that the total length of a text string is S bytes, the number of generated Rabin hash values is $(S-M+1)$. This amount of hashing makes CDC a CPU-bounded application. However, for content updates, CDC is more robust and flexible because only a few chunks close to the updated region need to be modified.

$$F_1 = (t_1 \cdot R^{M-1} + t_2 \cdot R^{M-2} + ... + t_M) \ mod \ Q \tag{4.1}$$

$$F_{i+1} = ((F_i - t_i \cdot R^{M-1}) \cdot R + t_{(i+M)}) \ mod \ Q \tag{4.2}$$

As summarized in Table 4.1, background experiments involving ten chunking policies with different average chunk size show diverse results in terms of chunking throughput and RE ratio. Our background testing environment is shown in Table 4.2 and uses only CPUs, not an FPGA, for computation. The 128-bit variant of *MurmurHash* [99] is applied to generate an FP for each chunk. We use a Bloom Filter (BF) as the space-efficient indexing data structure [101]. A BF has a controllable and acceptable false positive rate. The drawback of false positives is that some chunks are actually

Table 4.2: Composition of our testing environment.

	Component	Description
Hardware components	CPU	2×Intel E5-2670 @ 2.60 GHz
	RAM	DDR SDRAM/64 GiB
	DISK	2×2 TB 4K-stripe RAID-0 Flash based SSD
	NIC	2× 10-gigabit Ethernet NIC
NRE processing components	FP	MurmurHash 128 bits
	Indexing	Bloom Filter
	Caching	LRU-based

not duplicates but the index reports they are. Our solution is to record FPs of those misguided chunks to guarantee further fetches from the server side while setting the false positive rate to be extremely low, such as 0.025%. Since the size of each chunk generated by CDC may be variable, we use a chunk-based LRU caching policy in the client. That is, when a new data chunk is to be stored, the least recently used data chunk will be removed from the cache. The cache size is set to 32 GB. MODP and MAXP, both of which are IP packet-level NRE solutions, are discussed in [20, 23]. We specify the same sliding window size $M = 32$ bytes and sampling period $p = 32$ as the previous work [20]. The CDC policy is adopted from the solution of LBFS [25]. The tuple of $\langle min, avg, max \rangle$ is used to decide the minimum, expected, and maximum chunk size which satisfies $max = 4 \cdot avg = 16 \cdot min$ and is used as the default configuration. The *NFSv4* server contains 17.6 TB of files including collected *HTTP* text data, *AVI* video files, and *Linux images*. Tens of simulated clients request these files following a Zipf-like distribution.

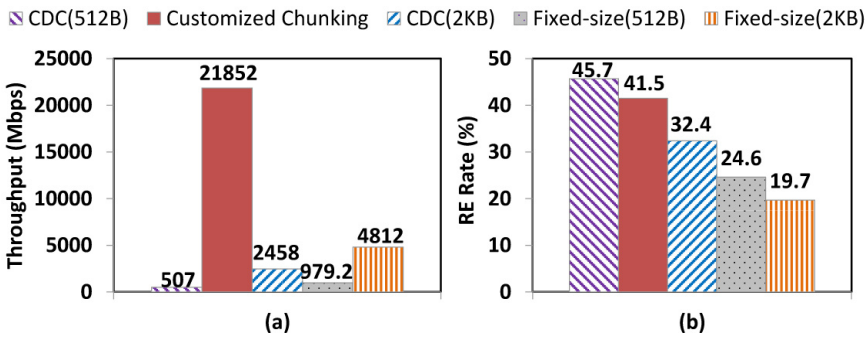


Figure 4.2: Throughput and RE ratio (%) in an *NFSv4* file server.

Table 4.3: An example of customized chunking policy.

Data Type	Chunking policy	Expected chunk Size
HTTP	CDC	1 KB
AVI video	fixed-size	32 KB
Linux images	fixed-size	8 KB

From Table 4.1, we can observe:

(i) The chunk size determines chunking throughput and RE ratio. Usually, a chunking policy with a smaller chunk size guarantees high RE ratio but sacrifices throughput.

(ii) Compared with fixed-size chunking, CDC with the same expected chunk size improves RE ratio by 9.5% to 16.4% but degrades throughput by 50% or more.

(iii) MODP and MAXP can achieve the best RE ratio but have the worst throughput (only tens of Mbps). Unfortunately, the throughput requirement of most MBs is on the order of Gbps [26], and such IP-level NRE solutions are unable to tune the expected chunk size to achieve both high RE ratio and adequate system throughput.

We conclude that CDC with a reasonable expected chunk size is capable of achieving both high RE ratio and throughput. Moreover, we can speed up the throughput of CDC using an FPGA accelerator without compromising RE ratio.

4.2.3 Flexibility in Configuring Chunking Policy

Many vendors developed their WOAs as NRE MBs preferentially operating in the TCP layer [87, 88]. Woo et al. [26] suggested that TCP-based solutions are more suitable than IP packet-based solutions for NRE. At the IP packet level, the NRE process is slowed down due to the processing burden of extremely small chunks. Another merit of a TCP-based NRE solution is that we can differentiate the chunking policy on a TCP flow basis. It has been well studied that various network flows show different degrees of redundancy [20, 26, 91].

We use different chunking policies to do a set of RE sampling and select a proper one for each data set as shown in Table 4.3. We call this approach customized chunking policy and it considers both the RE ratio and computation cost on a TCP flow basis. With the same experimental environment in Table 4.2, Figure 4.2 shows the throughput and RE ratio of the customized policy and four sample chunking polices from Table 4.1.

As shown in Figure 4.2, the throughput of the customized chunking policy outperforms the other four policies by 4.5X to 43X. Moreover, the RE ratio is only a little less than that of CDC with a 512 byte expected chunk size and outperforms the others. The major reasons are as follows:

(i) Nearly all redundant data often belongs to the same application rather than scattered across different applications.

(ii) CDC performs well for frequently modified and small-size applications, such as text-based HTTP files, while the fixed-size chunking policy is always a better choice than CDC for mostly read and large data size applications, such as virtual machine images and AVI video.

(iii) The chunking policy with a more coarse-granularity chunk size can dramatically reduce the demand on resources without losing much RE ratio for a mostly read application.

In short, when tailored to the redundancy characteristics of each TCP flow, a flexibly configured chunking policy with a suitable customized average chunk size can improve both network throughput and RE ratio. VNRE generates these optimized custom chunking policies automatically.

4.3 VNRE Design

Our VNRE design consists of three basic modules, and Figure 4.3 depicts their relationships. The VNRE Controller module imposes a customized chunking policy for each TCP flow and assigns the computation resources (i.e., either FPGA or CPU). The computation process is responsible for generating data chunks according to the configured chunking policy. The chunking throughput and RE ratio can be delivered to the monitor which uses this information to advise the controller to flexibly re-configure the chunking policy as further desired. The detailed design of the three basic models is presented in the following subchapters.

4.3.1 VNRE Controller

As shown in Figure 4.4, a flow table is built on the TCP socket layer to assist the VNRE controller to customize the chunking policy. Each item of the TCP flow table is

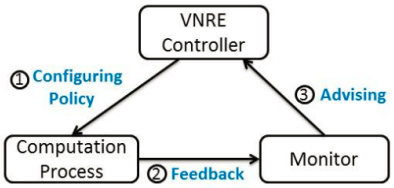


Figure 4.3: Three basic modules in VNRE.

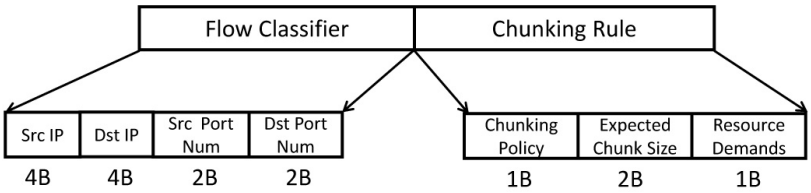


Figure 4.4: Data structure of the TCP flow table.

comprised of two elements: *Flow Classifier* and *Chunking Rule*.

The *Flow Classifier* differentiates TCP flows where the customized chunking policy is flexibly assigned. Specifically, we use four tuple elements, ⟨*Source (Src) IP, Destination (Dst) IP, Src Port Num, Dst Port Num*⟩, to identify TCP flows. The *Flow Classifier* occupies 12 Bytes. In the default configuration, *Src Port Num* specifies a concrete network application on the server side. We consider the source port number to be our primary measurement to classify TCP flows.

Chunking Rule is used to differentiate chunking policies on a TCP flow basis. A *Chunking Rule* contains three sub-parts: *Chunking Policy, Expected Chunk Size,* and *Resource Demands*. The *Chunking Policy* is initialized as CDC (state bits: "11"), fixed-size chunking (state bits: "01"), or no chunking (state bits: "00"). The *Expected Chunk Size* is initialized in the kilobyte range when a chunking policy has been configured. To customize a proper expected chunk size, in the beginning we use a warm-up process to decrease the chunk size until arriving at a certain RE ratio. The *Resource Demands* specifies the computation resource type, i.e., FPGA or CPU. If the fixed-size chunking policy is adopted, it is processed by the server's CPUs. For CDC, the number of PR-FPGA units is configured based on the performance requirement. In the current implementation, VNRE will assign the PRs equally to the flows which are in CDC mode.

4.3.2 Computation Process

FPGA Architecture

We propose an FPGA-based hardware acceleration solution for CDC as shown in Figure 4.5. As mentioned in Chapter 4.2.1, the PR-FPGA is used to quickly verify and evaluate multiple acceleration schemes. In this subchapter, we mainly present a generic FPGA-based scheme for the CDC process. The architecture primarily consists of six logic modules: a *PCI Express (PCIe) module*, an *Avalon Memory Mapped (Avalon-MM) to PCIe TLP (Transaction Layer Packet) bridge module*, a *DMA engine*, an *on-chip RAM* module, a *Rabin fingerprinting module*, and a *modulo module*. These modules are interconnected with each other by an Avalon-MM fabric. The role of each module is listed as follows:

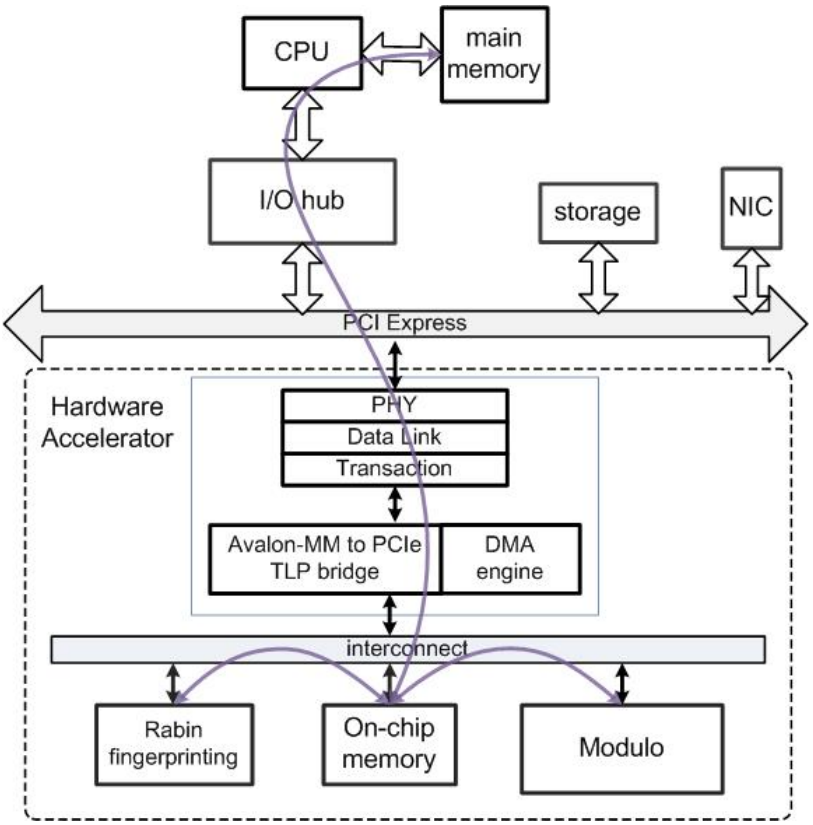


Figure 4.5: Architecture of the FPGA-based accelerator.

(1) The *PCI Express (PCIe)* module is used to interconnect upper-level commodity hardware components with the FPGA accelerator. The module is composed of three layers: a physical layer, a data link layer, and a transaction layer. The role of the transaction layer is using a packet as the basic I/O unit to communicate with all the computing components. As an intermediate stage, the data link layer is responsible for link management, data integrity, and link states. The physical layer is used to finish the underlying communication circuitry.

(2) The *Avalon-MM to PCIe TLP bridge* module connects the PCIe transaction layer to our user-defined CDC application layer. The bridge includes a protocol translator and a DMA engine. The role of the protocol translator is to interpret PCIe TLPs and then spawn semantically-conformable Avalon-MM transactions, and vice versa.

(3) The *DMA engine* moves data from the host memory to the accelerator's local memory, and vice versa. To improve the efficiency of RE, our design makes the DMA engine work in a chaining mode which leverages descriptors to specify source/destination addresses of data movement.

(4) The *on-chip RAM* is dual-ported and is used to fetch/store pending data into memory.

(5) The *Rabin fingerprinting* module is used to compute FPs of pending TCP flows. As shown in Figure 4.6, it consists of a central control logic, a bus interface logic, and some auxiliary registers. Based on the model of a Finite State Machine (FSM), the central control logic is designed to direct the computation of Rabin FPs step by step. The bus interface logic abides by the Avalon-MM specification and is used to interface the Rabin module with the on-chip bus. The auxiliary registers fulfill the functionalities of data storage (such as general data registers and Rabin FP registers), logic start or stop control commands, and status indication.

In addition, we take some measures to further optimize the performance of the Rabin fingerprint algorithm in terms of the analysis of the structure and properties of the algorithm. For example, as we previous mentioned, some constants involved in the fingerprinting are pre-calculated and stored into a recorded lookup table in a ROM logic. Thus, this module is finished by a group of logical units to finish two baseline operations: adding and bit shifting in Equation 4.2.

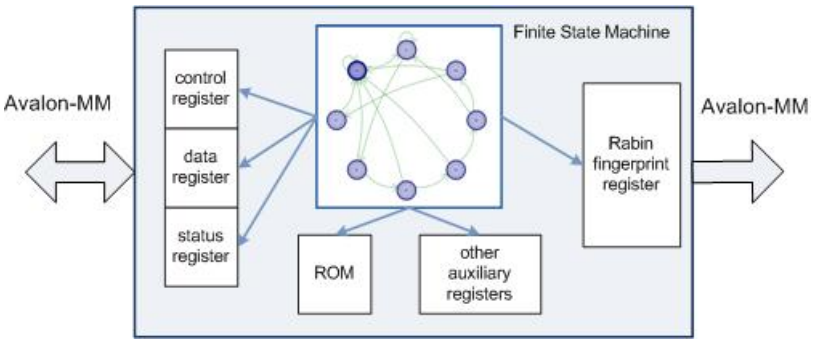


Figure 4.6: Rabin fingerprinting module.

Without acceleration, the traditional large number summation for hashing in Equation 4.1 costs much time and may have a larger margin of error. For example, using hexadecimal base 16 representation and a window size of 32, the equation's constants can generate exceedingly large numbers, like $(255 \cdot 16^{31})$, with values that exceed the upper limit of an *unsigned long* $(2^{64} - 1)$ or even a *float*. Thus, we take some measures to further optimize the performance of the Rabin FP calculation through analysis of the structure and properties of the algorithm. First, we calculate the 256 possible constants. Exploiting the addition property of modular arithmetic, the ith constant in the array table is computed as $(i \cdot 16^{31} \% Q)$, where "%" denotes the *mod* operation, and fits in the range of an *unsigned long* variable. The table involved in the fingerprinting is stored in a ROM logic block as a constant instantiated in the FPGA. Figure 4.7 demonstrates the FPGA computation process of Equation 4.1 and Equation 4.2. To get the result of Equation 4.1, we use the right shift operation ("$\gg$") and the constant table to quickly compute each value $V_{M-i} (1 \leq i \leq 32)$ in the FPGA. Each V_{M-i} is accumulated by the *Adder* operator, and then a modulus operation $(\%Q)$ obtains the final result F_1. As the sliding window shifts, we use Equation 4.2 to recursively calculate the subsequent FP value. This process includes the *Subtracter*, *Left shift* ("$\ll$"), *Adder*, and *Modulo* operations. Consequently, the Rabin FP computation process can be finished within a constant time by the single computation $(F_i \rightarrow F_{i+1})$.

(6) The *modulo* module is applied to conduct the modulo operations of CDC. Calculated by Equation 4.2, we have a FP value to logically *AND* in the FPGA with the average chunk size (e.g., 4 KB). When the result matches a predefined constant value

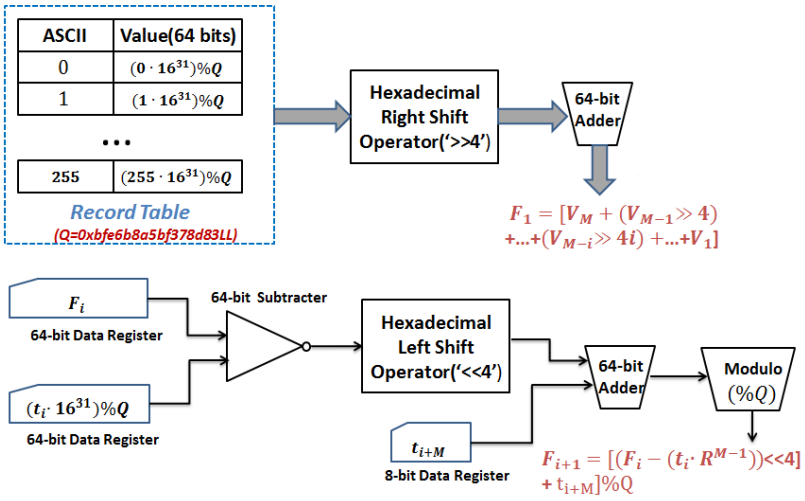


Figure 4.7: Rabin fingerprinting computation process. *R is set to 16 and M is set to 32 in our implementation. Q is initialized as a large 64-bit prime.*

(e.g., 0x01), we refer to the last bit of the Rabin window as a cut point. This is how a CDC scheme statistically decides its chunk boundary. However, this variability may bring some unexpected conditions. If the cut point is less than the minimal or greater than the maximal chunk size, the minimal or maximal chunking size is chosen as the final value, respectively.

CDC Operation

For convenience, we suppose that: 1) the Deduplication-Pending Data (DPD) has been stored in the host memory; 2) the descriptors used for DMA transfers have been populated in a descriptor table in the host memory to differentiate TCP flows, and the descriptor table is managed by the device driver; and 3) the accelerator initialization has been completed during the boot period. The process of RE encompasses three stages: Stage I aims to transfer DPD to the local I/O buffer, which is initialized as 256 KB. For a TCP flow, the device driver is used to establish the mapping relationship between its buffer and corresponding items in the descriptor table. Prior to the transfer of pending data blocks, the DMA engine copies the entire descriptor table into the local memory on the hardware accelerator under the guidance of the descriptor controller.

Each descriptor item includes source and destination addresses of a data transfer. Based on the descriptor information, the DMA engine transfers data between the host memory and the accelerator memory.

The data flow slicing is performed during the Stage II period. When the DMA engine completes the transfer of a pending data block, the upper-layer application can program the registers in the *Rabin fingerprinting* module to start computing the data FPs. According to the computation process in Equation 4.2, the Rabin fingerprinting module reads data from the local buffer with a regular-size window.

After completing the chunking of the whole data buffer, the acceleration goes to Stage III. The accelerator interrupts the CPU and then passes back the information about cut-points to the host memory via the DMA engine. As soon as one round finishes, the NRE process will launch a new round and iteratively fulfill the remaining pending data.

Algorithm 3 Rabin fingerprinting start control.

```
 1: always@(posedge clk or negedge reset)
 2: begin
 3: if (reset == 1'b0)
 4:     control_reg <= 0;
 5: else
 6:     begin
 7:     if (chipselect & write & control_reg_select)
 8:         begin
 9:         if (byteenable[0])
10:             control_reg[7 : 0] <= writedata[7 : 0];
11:         end if
12:         if (byteenable[1])
13:             control_reg[15 : 8] <= writedata[15 : 8];
14:         end if
15:         end
16:         ...
17:     end if
18:     end
19: end if
20: end
```

For the purpose of demonstration, we describe two representative blocks in the Rabin fingerprinting module's Verilog. When the three signals, *chipselect*, *write*, and *control_reg_select*, are valid simultaneously and the start-bit in the control register is set to 1 by the VNRE controller, the following deduplication operations of the pending data are swiftly launched once the state machine detects the set event. The procedure where the data on the Avalon-MM bus are written to the control register is depicted

Algorithm 4 Rabin fingerprinting lookup operation.

```
1:  always@(posedge clk or negedge reset)
2:  begin
3:  if (reset == 1'b0)
4:      tab_data_reg <= 64'b0;
5:  else
6:      begin
7:      if (nxt_state == S2)
8:          tab_data_reg <= mem[p1_data_reg[7 : 0]];
9:      end if
10:     end
11: end if
12: end
```

in Algorithm 3. Algorithm 4 illustrates that when prior operations are finished and the finite state machine is in State 2, the logic block uses the value in an intermediate register $p1\_data\_reg$ as an index to fetch a constant from the lookup table. Then, the stored value $p1\_data\_reg$ is assigned to another auxiliary register tab_data_reg for reuse on a rising edge of the clock.

4.3.3 Monitor

In the beginning, each flow can be configured with a chunking policy and the expected chunk size. To make VNRE adaptively adjust the parameters, we use a Monitor to capture the RE ratio and this kind of throughput. For some extreme cases, the throughput of a TCP flow becomes the bottleneck when the flow pattern changes. To satisfy the overall performance demand, we can make the expected chunk size larger. Conversely, when the RE ratio of a TCP flow cannot satisfy our expectation, we can make the expected chunk size smaller. For some flows, if the RE ratio is near zero, we can turn off NRE to maximize throughput.

The format of the chunking log file is shown in Figure 4.8. The Monitor records a line of metadata items for each TCP flow. The *time* item is formatted as the global Greenwich Mean Time (GMT) at the start of chunking. The *srcPort* is used to identify a TCP flow. The average throughput of a TCP flow in a period is calculated as $\sum(Data\ Size)/\sum(Processing\ Time)$ where all the items have the same *srcPort*. The RE ratio is calculated as $1- \sum(Sending\ Data\ Size)/\sum(Data\ Size)$ where *Sending Data Size* is the amount of data sent over the network after RE and *Data Size* is the original amount of data requested. At the end of a constant period (e.g., 10 minutes), the Monitor will

Time	srcPort	Data Size	Processing Time	Sending Data Size

Figure 4.8: Format of the chunking log file.

calculate and send the RE ratio and the average throughput of each flow to the VNRE Controller, which will decide the reconfiguration polices.

4.4 VNRE Evaluation

We consider three main aspects in the evaluation: 1) measuring the speedup of CDC throughput due to the FPGA accelerator, 2) determining the improvement in both RE ratio and throughput through our VNRE controller, and 3) analyzing overhead when adding a NRE service.

Our experimental hardware platform is built on a *Terasic DE5-Net Stratix V GX FPGA* [102] in a standard x86 server. The hardware resources including CPUs, memory, disks, and the FPGA are managed by OpenStack in OpenANFV[27]. A Virtual Machine (VM) can access a certain number of PRs via the Single Root I/O Virtualization (SR-IOV) technique. Our baseline testing components are described in Table 4.2.

4.4.1 Speedup Ratio of CDC Throughput by Using FPGA Accelerator

Data Set Description. About 320 files including metadata, string-texts, videos, and Linux images are sampled. The file size ranges from 0.4 KB to 1.96 GB. When a file is less than the predefined minimum chunk size, this file will not be processed by CDC. The average file size is 15.6 MB and the total size of the data set is about 4.87 GB.

To evaluate the performance in terms of hardware speedup, we test CDC by assigning four distinct computing configurations. The expected chunk sizes of CDC are initialized as 512 bytes, 1 KB, and 2 KB. From the results shown in Figure 4.9, we can observe:

(1) Compared to assigning one virtual CPU (vCPU), the throughput of assigning one PR to handle the CDC process is improved by 2.7X-2.8X. For example, when the expected chunk size is 512 bytes, the throughput of CDC using one PR is 200.6 Mbps; while using one vCPU the throughput only reaches 70.7 Mbps.

(2) CDC throughput grows nearly linearly when the number of PRs for processing

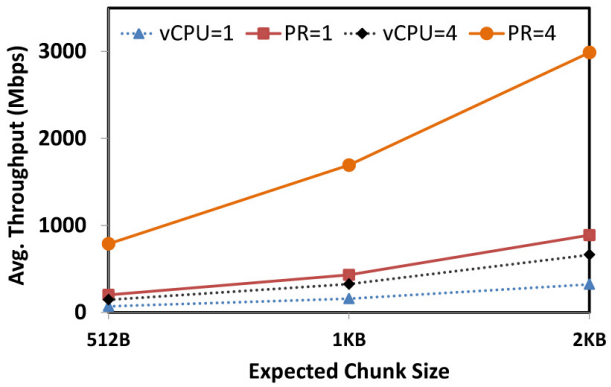


Figure 4.9: Average CDC throughput with FPGA Accelerator (PR) and without (vCPU).

CDC is increased linearly. The growth trend of CDC throughput is slower when the number of vCPUs processing CDC is increased linearly. For example, when the expected chunk size is 1KB, CDC throughput using four PRs is 1692.7 Mbps, nearly fourfold that when using one PR (430.8 Mbps). The throughput using four vCPUs is 324.8 Mbps, while using one vCPU can only provide 156.9 Mbps. Consequently, in our platform, the CDC throughput obtained by using one PR is still faster than that of using four vCPUs.

CDC handled by the FPGA can alleviate the server's CPU burden since both CDC and the chunk's FP computation are CPU-bound. As shown in Figure 4.10(a), we compare the accumulated throughput when both MurmurHash and CDC are computed by one vCPU to that of when one PR is added to offload CDC processing. When MurmurHash is computed by one vCPU and CDC is computed by one PR unit, the accumulated throughput is improved by 6.3X. The throughput of CDC has improved 9.5X and the MurmurHash throughput speedup is 4.3X. As shown in Figure 4.10(b), when the number of vCPUs is four, offloading CDC to one PR unit improves the accumulated throughput 4.9X. The CDC throughput speedup is 5.0X, and the throughput of MurmurHash is improved 4.8X.

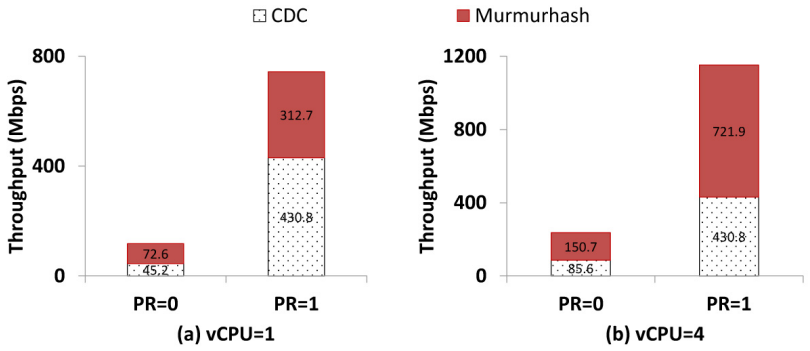


Figure 4.10: Accumulated throughput of CDC and 128-bit MurmurHash for FP generation with FPGA CDC offloading (PR=1) and without (PR=0).

Table 4.4: Description of four TCP flows.

Application	Server Port	Data Size
HTTP	80	3.6 TB
RTSP	554	1.2 TB
SMTP	25	0.6 TB
FTP-data	20	2.4 TB

4.4.2 Improvements from Flexibly Configured Chunking Policy

Testing Environment Description. As shown in Table 4.4, we collect four typical data sets. We set up an FTP server that only contains Linux images. The CDC process is performed by the FPGA substrate. The number of PRs is limited to four in our testing. Both fixed-size chunking and 128-bit MurmurHash are calculated by vCPUs. The number of vCPUs is also set to four. The interval of sending results from the Monitor module to the VNRE Controller module is set to 10 minutes when using VNRE.

Figure 4.11 shows an overview of the experimental configuration. The first procedure is to simulate the client network request behavior. There are four kinds of clients that generate network data access requests: HTTP client, RTSP client, SMTP client, and FTP client. Each client access pattern obeys a Zipf-like distribution [103] where the ranking argument is set to 0.99. The generating period lasts for three days. Moreover, each flow has a diurnal pattern with distinguishable light and heavy load periods to simulate daily access trends. The local cache size of the MB proxy for indexing and

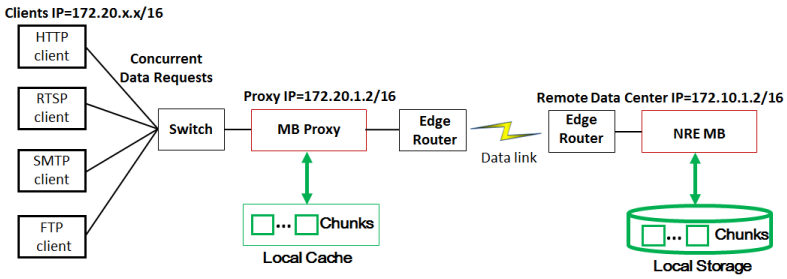


Figure 4.11: Overview of the experimental configuration.

storing chunks is set to 32 GB. The data link and its infrastructure (switches and routers) support gigabit Ethernet. Both CDC and FP generation processes are handled by the NRE MB on the remote data center side. All the network files are chunked inline and stored in a backup database storage system.

A comparison of our TCP flow-based VNRE to four fixed-policy approaches in terms of NRE throughput and RE ratio is shown in Figure 4.12. The two CDC and two fixed-size chunking configurations, as well as our VNRE customized scheme when it is in a CDC mode, use FPGA acceleration. The throughput of the VNRE customized chunking policy is 6X to 57X that of the four predefined policies. This substantial increase is because that VNRE can select proper chunking sizes for different flows based on the feedback of the Monitor. Based on the port number, VNRE will select HTTP and SMTP flows to work in CDC mode while the other two work in fixed-size mode. After VNRE decides the basic chunking policy for each flow, it will assign the PRs to the flows which are in CDC mode. After running for a period of time, the expected chunk sizes are tuned to 512 bytes for HTTP, 16 KB for RTSP, 4 KB for SMTP, and 8 KB for FTP-data. Regarding RE ratio, the percent reduction from VNRE is only a little less, about 4.2%, than that of the CDC policy with a 512 byte expected chunk size, and our method beats the other three policies by 7.9% to 30.2%. In summary, varying the chunking policy and the expected chunk size based on the TCP flow in VNRE can dramatically improve chunking throughput while maintaining a high RE ratio.

Figure 4.13 demonstrates the average throughput improvement for client requests when using VNRE compared to no NRE of any kind. Measured at 3-hour intervals over a 72-hour period, the average throughput improvement due to VNRE is 1.4X to 6.5X.

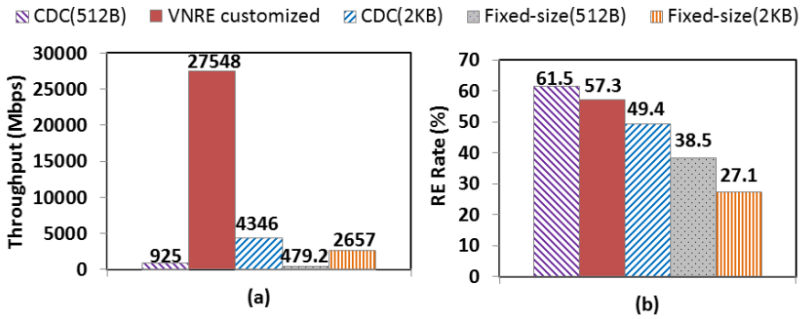


Figure 4.12: NRE throughput and RE ratio (%) of four sample chunking policies compared to customized VNRE control.

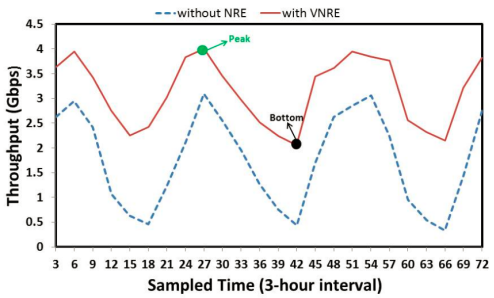


Figure 4.13: Comparison of expected throughput over time for client requests with VNRE and without using VNRE.

This faster response time shows that VNRE allows clients to get the requested content with less latency. Moreover, VNRE flattens the throughput curve. As labeled in Figure 4.13, the difference between "peak" throughput and "bottom" throughput using VNRE is smaller than without using NRE. This flattening occurs because VNRE is able to provide an even better throughput improvement during periods of fewer client requests.

To further analyze the effect on RE ratio of chunking policy, we compare the RE ratio of four TCP flows using three chunking polices and summarize the results in Figure 4.14. CDC and fixed-size are configured as shown in Table 4.4, while "Optimal" is a CDC with an unlimited chunk cache for a theoretical reference. Compared to fixed-size chunking, CDC has a much better RE ratio in the HTTP and SMTP traces but outperforms less obviously in RTSP and FTP-data. This result is because the RE ratio depends on the access behavior and degree of redundancy of different applications. For

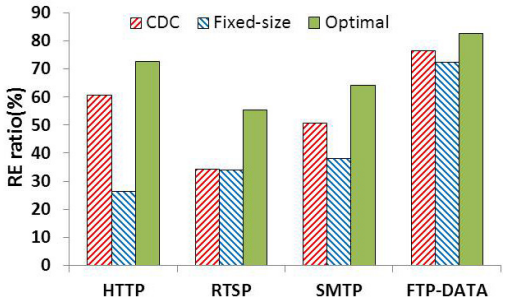


Figure 4.14: RE ratio (%) of four TCP protocols using three chunking polices.

example, the tested HTTP files are mostly writes and have more redundant chunks. The RTSP files are mostly reads and share few repeated chunks in different files. RE ratios are high when clients access the same files within a certain period of time, but only the "Optimal" CDC with its hypothetical unlimited chunk cache can retain a high RE ratio over longer periods between same-file access.

4.4.3 Overhead Analysis

According to our experimental setup, the acceleration system spends about 2000 clock cycles, on average, including around 1500 cycles transferring a 4 KB page from the system memory to the local memory, roughly 310 cycles passing back the chunking results, and approximately 190 cycles for the acceleration algorithm itself. The average processing latency of a packet due to the VNRE process is 11.5 μs. In the worst case, the latency is up to 25 μs. However, as shown in Figure 4.13, clients get responses for the requested content with much less latency when using the VNRE process. The descriptor table occupies on the order of tens of kilobytes in main memory, and a data buffer of 4 MB is sufficient.

4.5 OpenANFV: Accelerating Network Function Virtualization with a Consolidated Framework in OpenStack

As aforementioned, we use the FPGA accelerator to speed up NRE process. This subchapter will extend the above work and use such dedicated hardware for multiple

functions. Specified appliances or *middleboxes* (MBs) have been explosively used to satisfy a various set of functions in operational modern networks, such as enhancing security (e.g. firewalls), improving performance (e.g. WAN optimized accelerators), providing QoS (e.g. Deep Packet Inspection (DPI)), and meeting the requisite others [104]. Network Function Virtualization (NFV) recently has been proposed to optimize the deployment of multiple network functions through shifting the MB processing from customized MBs to software-controlled inexpensive and commonly used hardware platforms (e.g. Intel standard x86 servers) [105]. However, for some functions (e.g. DPI and Network Deduplication (Dedup) or NRE, Network Address Translation (NAT)), the commodity shared hardware substrate remain limited performance. For a standard software based Dedup MB (Intel E5645, 2.4GHZ, 6 cores, exclusive mode), we can only achieve 267Mbps throughput in each core at most. Therefore, the resources of dedicated accelerators (e.g. FPGA) are still required to bridge the gap between software-based MB and the commodity hardware.

To consolidate various hardware resources in an elastic, programmable and reconfigurable manner, we design and build a flexible and consolidated framework, OpenANFV, to support virtualized acceleration for MBs in the cloud environment. OpenANFV is seamlessly and efficiently put into Openstack to provide high performance on top of commodity hardware to cope with various virtual function requirements. OpenANFV works as an independent component to manage and virtualize the acceleration resources (e.g. *cinder* manages block storage resources and *nova* manages computing resources). Specially, OpenANFV mainly has the following three features.

- **Automated Management.** Provisioning for multiple VNFs is automated to meet the dynamic requirements of NFV environment. Such automation alleviates the time pressure of the complicated provisioning and configuration as well as reduces the probability of manually induced configuration errors.

- **Elasticity.** VNFs are created, migrated, and destroyed on demand in real time. The reconfigurable hardware resources in pool can rapidly and flexibly offload the corresponding services to the accelerator platform in the dynamic NFV environment.

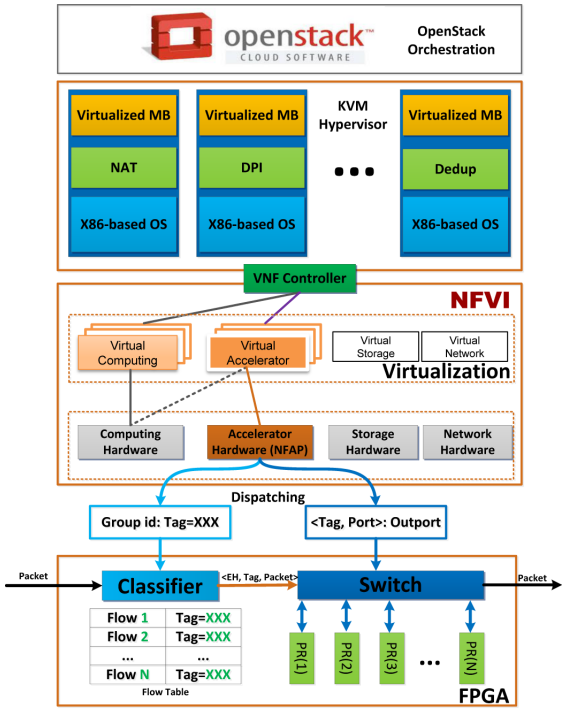


Figure 4.15: Brief OpenANFV architecture.

- **Coordinating with OpenStack.** The design and implementation of the OpenANFV APIs coordinate with the mechanisms in OpenStack to support required virtualized MBs for multiple tenancies.

4.5.1 Architecture and Implementation

Figure 4.15 briefly shows the architecture of OpenANFV. From the top-to-down prospective view, when a specific MB is needed, its required resources are orchestrated by OpenStack. There are sufficient northbound APIs in this open platform. Each VNF of MB is instantiated by a Virtual Machine (VM) node, running on the common x86 platform. Once deployed, these MBs aim to provide services as well as the original appliances. The role of *VNF controller* is to leverage the resource demand of VMs associated with underlying virtual resource pools.

NFV Infrastructure (NFVI) comes from the concept of *IaaS* to virtualize corresponding hardware resources. In NFVI, the *Network Functions Acceleration Platform*

(NFAP) provides a heterogeneous PCIe based FPGA card which supports isolated *Partial Reconfigure* (PR) [106]. PR can virtualize and reconfigure the acceleration functions shortly in the same FPGA without affecting the exiting virtualized accelerators. Using PR, different accelerators are placed and routed in special regions.

The FPGA is divided into static and partially reconfigurable regions (PR). When the network requirements are changed, the function of PR could be replaced with a new one. The PR is implemented to support different accelerators, and controlled by one VM exclusively. When one accelerator is reloading without affecting the other accelerators are not affected. The PR and VM can communicate through PCIe SR-IOV efficiently. The static region contains the shared resources (e.g. the storage and network interfaces).

The responsibility of NFAP is dispatching the rules to the two modules of FPGA, the *classifier* and the *switch*. The *classifier* identifies the flow where the rule format conforms to $\langle$*group id, tag*$\rangle$, when the flow matches the condition *group id*, classifier will add an item to the flow table in the classifier. Based on the flow table, the classifier could divide the traffic into different NFs. The switch get the MB chain configuration [$\langle$*Tag,Port*$\rangle$:*Outport*], and forward the encapsulated head (EH) based packet combining with the tag and income port. The packets are redirected to the MB in the chain in sequence until to the last one. The encapsulated packet with its tag can transfer more than one PRs, moreover, the tag could also support the load balancing between the NFs. Compared with FlowTags [107], we use the tag in the encapsulation to expand flexibly without affecting the field of the original packet header.

We have implemented the prototype of integrating NFAP into OpenStack following the methodology which is proposed by the ETSI standards group [108]. The K-V pair of <*vAccelerator*, *number of running vAccelerators*> is ued to track the current status of NFAP. *vAccelerator* is the aliased key to identify the PCIe device and *number of running vAccelerators* is to identify the current virtual accelerators. The scheduler has been finished in the *VNF controller* which is followed by the standard *nova scheduler*. Finally, the extended *VM.xml generation* includes allocation of a PCIe device virtual function to a VM.

4.5.2 OpenANFV Evaluation

We evaluate the prototype of NFAP in a demo FPGA cards (Altera FPGA Stratix A7, 8GB DDR3, 8M QDR, 4 SPF+). The experimental environment consists of one x86-based server (2.3GHz 8Core Intel Xeon E5, 96GB RAM, 40G NIC) with Altera FPGA card, two x86-based servers running the controller and OpenStack, respectively. The IXIA XM2(with NP8 board) is used as the source and the sink of packets. The server with the NFAP runs KVM hypervisor, and each NFAP could provide three empty PRs for the server. The VMs used for the test have the same configuration (1 vCPU, 16G RAM and 1TB Disk).

Our tests include three VNFs, NAT, DPI, and Dedup. Each VNF has two versions, with and without adopting NFAP. Without NFAP, the computing resources of VNFs are completely provided by *nova* in OpenStack. For NFAP assisted NAT, the software in the VM has an Openflow-like API with the NFV controller and configures the policy using a hardware abstract API. The hardware part in the NFAP offloads the flow table, the header replacement, and the checksum calculation from the VM. The packet will be merely processed in the NFAP, if the flow is matched in the flow table. For NFAP assisted DPI, we offload all the string match (Mutihash Algorithm and Bloomfilter), regular match, and the rules table in the NFAP, and the VM keeps the rule compiler and the statistics which are needed by the controller. The rule compiler compiles perl compatible regular expression to Deterministic Finite Automata (DFA) rule. For NFAP assisted Dedup, we offload the rabin hash, Marker select algorithm and chunk hash (Murmur hash). As the TCP is too complex to implement in the FPGA, the tcp-stack is still in the VM and the packets are received and sent out via the software tcp-stack. We still do some optimizations like using the Data Plane Development Kit (DPDK) driver and use space tcp-stack. As shown in Figure 4.16, the performance of DPI, Dedup, and NAT with adopting NFAP in OpenANFV outperforms the scheme without NFAP by 20X, 8.2X, and 10X, respectively.

4.6 Related Work

VNRE builds upon the following previous work.

Network Redundancy Elimination. Various network systems are well designed

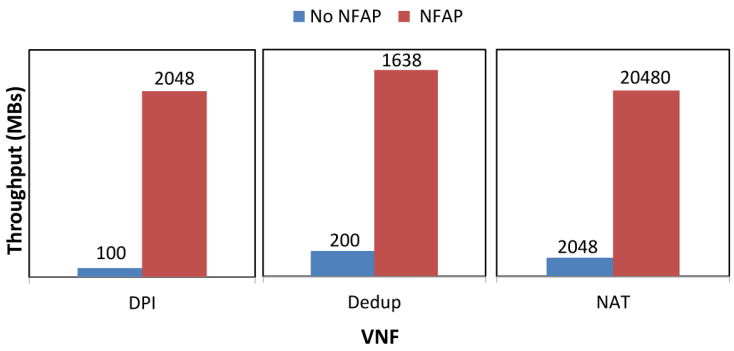


Figure 4.16: Performance results on throughput with or without adopting NFAP.

for eliminating redundancy for WAN links. Some of them operate at the object or application levels, such as web proxy servers [109]. In recent years, protocol-independent NRE services have been developed to provide better redundancy elimination [20]. This approach was first pioneered by Spring et al. [21], and later many network service vendors developed their WOAs as MBs as we previously mentioned. IP packet-level NRE solutions like MODP and MAXP [20, 23] are unable to tune the expected chunk size to achieve both high RE ratio and system throughput. LBFS [25] is a network file system designed for low-bandwidth networks and uses a typical NRE solution to reduce traffic burdens. Cloud computing emerges in recent years [110, 111]. Zohar et al. presented a novel receiver-based end-to-end NRE mechanism, namely PACK, to eliminate redundant traffic between the cloud and its end-users [112]. CloudNet [113] used a content-based NRE solution to eliminate the redundant data while transferring VM memory and disk state in a virtualized cloud system. Research on NRE solutions in WOAs is well studied on common x86-based platforms [23, 26, 95]. NRE has also been studied in network backup systems [114] and in Software-Defined Networks [115].

Network Function Acceleration by FPGA. NetFPGA [93] provides a standard interface among FPGA stages. In our work, an internal NetFPGA is managed by the VNRE controller. As designed for high-performance and scalable networks, NetFPGA has been applied to many virtual network infrastructures such as virtual routers [116], virtual network data planes [94], and virtual switches [117]. The NetFPGA-based PR-FPGA substrate has been proposed in various virtualized cloud platforms [98].

Network Function Virtualization Network Function Virtualization (NFV) becomes a prevalent trend in cloud computing. A variety of virtualized functions are built in a shared hardware resources including FPGA substrate through centralized software-controlled pooling management [118]. OpenNF [119] builds a consolidated control plane to manage both network forwarding state and internal NFV states. Software-based control for MB platform has been studied extensively [120–122]. The Arrakis OS [123] supports hardware I/O Virtualization, whose kernel is operated as the control plane, dynamically configuring the data path to each application including NFVs. ClickOS [124] rapidly manipulates a variety of NFVs in a software controlling manner on inexpensive, commodity hardware (e.g., x86 servers with 10Gb NICs).

4.7 Conclusion

In summary, this chapter presents a specially-designed NRE appliance with a unique intelligent controller and FPGA-based acceleration to provide improved throughput and RE ratio. The VNRE controller is designed to tune the chunking policy and its expected chunk size on a TCP flow basis. Compared with an empirical chunk size, adjusting chunking policies to accommodate the features of each flow shows exceptional improvement in throughput. We validate that the throughput of client requests can be greatly improved with VNRE versus a selection of static configurations. Moreover, we propose an improved FPGA-based scheme to speed up CDC while retaining its high RE ratio. The FPGA Verilog code of our CDC accelerator (as well as a MurmurHash accelerator not discussed in this work) is openly available for future study at `https://github.com/xiongzige/VNRE`. Furthermore, to consolidate various hardware resources in an elastic, programmable and reconfigurable manner, we design and build a flexible and consolidated framework, OpenANFV, to support virtualized acceleration for MBs in the cloud environment.

Chapter 5

Conclusion

Large volumes of data being continuously generated drive the emergence of large capacity high performance storage systems. Recently more new storage devices/technologies have emerged. To reduce the total cost of ownership, storage systems are built in a more composite way incorporating the emerging storage technologies/devices, including Storage Class Memory (SCM), Solid State Drives (SSD), Shingle Magnetic Recording (SMR) and even across off-premise cloud storage. This makes enterprise storage hierarchies more interesting and diverse. To make better utilization of each type of storage, industries have provided multi-tier storage through dynamically placing hot data in the faster tiers and cold data in the slower tiers. Data movement happens between devices on one single device and as well as between devices connected via various networks. This thesis aims to improve data management and data movement efficiency in such hybrid storage systems.

To bridge the giant semantic gap between applications and modern storage systems, passing a piece of tiny and useful information (I/O access hints) from upper layers to the block storage layer may greatly improve application performance or ease data management in storage systems. This is especially true for heterogeneous storage systems. Since ingesting external access hints will likely involve laborious modifications of legacy I/O stacks, thus making it is very hard to evaluate the effect of access hints.

This thesis presents and develops a generic and flexible framework, called HintStor, to quickly play with a set of access hints and evaluate their impacts on heterogeneous storage systems. The design of HintStor contains a new application/user level interface,

a file system plugin and a block storage data manager. With HintStor, storage systems composed of various storage devices can perform pre-devised data placement, space reallocation and data migration polices assisted by the added access hints. HintStor supports hints either statically extracted from the existing components (e.g. internal file system data structure) or defined and configured by the users (e.g. streaming classification). We demonstrate the flexibility of HintStor by evaluating three types of access hints: file system data classification, stream ID and cloud prefetch on a Linux platform. The results show that HintStor is able to execute and evaluate various I/O access hints under different scenarios with minor modifications to the kernel and applications.

Industries have provided multi-tier storage through dynamically placing hot data in the faster tier and cold data in the slower tier. However, each kind of storage device/technology has its own unique price-performance tradeoffs and idiosyncrasies with respect to workload characteristics they prefer to support. Moving data tier by tier may not be efficient and even worse it may lead to unnecessary data movements.

This thesis studies the storage architecture with fully connected (i.e., data can move from one device to any other device instead of moving tier by tier) differential pools (each pool consists of storage devices of a particular type) to suit diverse types of workloads. To explore the internal access patterns and thus efficiently place data in such a fully connected topology, we propose a chunk-level storage-aware workload analyzer framework, simplified as ChewAnalyzer. Access patterns are characterized as a collective I/O accesses in a chunk composed of a set of consecutive data blocks. The taxonomy rules are defined in a flexible manner to assist detecting chunk access patterns. In particular, ChewAnalyzer employs a Hierarchical Classifier to exploit the chunk patterns step by step. In each classification step, the chunk placement recommender advises new data placement policies according to the device properties. Both pool status and device properties are considered in making placement decisions. ChewAnalyzer++ is designed to enhance the workload profiling accuracy by partitioning selective chunks and zooming in their interior characteristics. According to the analysis of access pattern changes, the storage manager can adequately distribute the data chunks across different storage pools. ChewAnalyzer improves initial data placement and migrations of data into the proper pools directly and efficiently if needed. We build our prototype for a storage system composed of Storage Class Memory (SCM), Solid State Drive (SSD) and Hard

Disk Drive (HDD) in a Linux platform. Through trace driven approach, our experimental results show ChewAnalyzer outperforms the conventional dynamical tiering by less latency and less write times on the flash pool. The total amount of data being migrated is also reduced.

To reduce the duplicate content transferred between local storage devices and devices in remote data centers, NRE aims to improve network performance by identifying and removing repeated transmission of duplicate content from remote servers. Using a CDC policy, an inline NRE process can obtain a higher RE ratio but may suffer from a considerably higher computational requirement than fixed-size chunking. Additionally, the existing work on NRE is either based on IP packet level redundancy elimination or rigidly adopting a CDC policy with a static empirically-decided expected chunk size. These approaches make it difficult for conventional NRE MiddleBoxes to achieve both high network throughput to match the increasing line speeds and a high RE ratio at the same time.

This thesis presents a design and implementation of an inline NRE appliance which incorporates an improved FPGA-based scheme to speed up CDC processing to match the ever increasing network line speeds while simultaneously obtaining a high RE ratio. The overhead of Rabin fingerprinting, which is a key component of CDC, is greatly reduced through the use of a record table and registers in the FPGA. To efficiently utilize the hardware resources, the whole NRE process is handled by a VNRE controller. The uniqueness of this VNRE that we developed lies in its ability to exploit the redundancy patterns of different TCP flows and customize the chunking process to achieve a higher RE ratio. VNRE will first decide if the chunking policy should be either fixed-size chunking or CDC. Then VNRE decides the expected chunk size for the corresponding chunking policy based on the TCP flow patterns. Implemented in a partially reconfigurable FPGA card, our trace driven evaluation demonstrates that the chunking throughput for CDC in one FPGA processing unit outperforms chunking running in a virtual CPU by nearly 3X. Moreover, through the differentiation of chunking policies for each flow, the overall throughput of the VNRE appliance outperforms one with static NRE configurations by 6X to 57X while still guaranteeing a high RE ratio. In addition, to consolidate various hardware resources in an elastic, programmable and reconfigurable manner, we design and build a flexible and consolidated

framework, OpenANFV, to support virtualized acceleration for MBs in the cloud environment. OpenANFV is seamlessly and efficiently put into Openstack to provide high performance on top of commodity hardware to cope with various virtual function requirements. OpenANFV works as an independent component to manage and virtualize the acceleration resources.

References

[1] Enterprise capacity 3.5 hdd (helium). http://www.seagate.com/enterprise-storage/hard-disk-drives/enterprise-capacity-3-5-hdd-10tb/.

[2] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating host aware smr drives. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, 2016.

[3] Jim Gray and Bob Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, 2008.

[4] Ahmadshah Waizy, Dieter Kasper, Jürgen Schrage, Bernhard Höppner, Felix Salfner, Henning Schmitz, Joos-Hendrik Böse, and SAP Innovation Center Potsdam. Storage class memory evaluation for sap hana. *HPI Future SOC Lab: Proceedings 2013*, 88:63, 2015.

[5] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–11. IEEE, 2014.

[6] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. Non-volatile storage. *Communications of the ACM*, 59(1):56–63, 2015.

[7] Latency of cloud storage. http://www.securstore.com/blog/latency-of-cloud-storage/.

[8] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *NSDI*, pages 543–557, 2015.

[9] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *FAST*, pages 315–322, 2016.

[10] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[11] Yi Liu, Xiongzi Ge, Xiaoxia Huang, and David HC Du. Molar: A cost-efficient, high-performance ssd-based hybrid storage cache. *The Computer Journal*, page bxu156, 2015.

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

[13] Hui Wang and Peter J Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST*, pages 229–242, 2014.

[14] Fabricpool preview: Building a bridge from the all-flash data center to the clouds. `https://community.netapp.com/t5/Technology/FabricPool-Preview-Building-a-Bridge-from-the-All-Flash-Data-Center-to-the/ba-p/124388/`.

[15] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Storage workload isolation via tier warming: How models can help. In *ICAC*, pages 1–11, 2014.

[16] Yue Cheng, M Safdar Iqbal, Aayush Gupta, and Ali R Butt. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 45–56. ACM, 2015.

[17] Jorge Guerra, Himabindu Pucha, Joseph S Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, volume 11, pages 20–20, 2011.

[18] Qiao Li, Liang Shi, Chun Jason Xue, Kaijie Wu, Cheng Ji, Qingfeng Zhuge, and Edwin Hsing-Mean Sha. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *FAST*, pages 125–132, 2016.

[19] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1683–1694. ACM, 2015.

[20] Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. Redundancy in network traffic: findings and implications. In *Proceedings of the Sigmetrics'09*, pages 37–48. ACM, 2009.

[21] Neil T Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 87–95. ACM, 2000.

[22] Xiongzi Ge, Yi Liu, Chengtao Lu, Jim Diehl, David HC Du, Liang Zhang, and Jian Chen. Vnre: Flexible and efficient acceleration for network redundancy elimination. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 83–92. IEEE, 2016.

[23] Bhavish Agarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. Endre: An end-system redundancy elimination service for enterprises. In *NSDI*, pages 419–432, 2010.

[24] Udi Manber et al. Finding similar files in a large file system. In *Usenix Winter*, volume 94, pages 1–10, 1994.

[25] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[26] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 319–332. ACM, 2013.

[27] Xiongzi Ge, Yi Liu, David HC Du, Liang Zhang, Hongguang Guan, Jian Chen, Yuping Zhao, and Xinyu Hu. Openanfv: accelerating network function virtualization with a consolidated framework in openstack. In *Proceedings of the SIGCOMM 2014*, pages 353–354. ACM, 2014.

[28] Western digital teases 14tb ultrastar hard drive. `https://www.bit-tech.net/news/hardware/2016/12/07/wd-14tb-ultrastar/1`.

[29] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 457–473. ACM, 2015.

[30] Scsi storage interfaces. `http://www.t10.org/`.

[31] Fay Chang and Garth A Gibson. Automatic i/o hint generation through speculative execution. In *OSDI*, volume 99, pages 1–14, 1999.

[32] fadvise(2). `https://linux.die.net/man/2/fadvise`.

[33] ionice(1). `https://linux.die.net/man/1/ionice`.

[34] A general purpose, write-back block cache for linux. `https://github.com/facebookarchive/flashcache`.

[35] bcache. `https://www.kernel.org/doc/Documentation/bcache.txt`.

[36] btrfs. `https://btrfs.wiki.kernel.org`.

[37] Device-mapper resource page. `https://www.sourceware.org/dm/`.

[38] kcopyd. `https://www.kernel.org/doc/Documentation/device-mapper/kcopyd.txt`.

[39] Lvm2 resource page. `https://sourceware.org/lvm2/`.

[40] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware.* " O'Reilly Media, Inc.", 2005.

[41] Technical committee t13 at attachment. `http://www.t13.org/`.

[42] Hdfs users guide. `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html`.

[43] kafka:a distributed streaming platform. `http://kafka.apache.org/`.

[44] P Daniel, Cesati Marco, et al. Understanding the linux kernel, 2007.

[45] Fiemap ioctl. `https://www.kernel.org/doc/Documentation/filesystems/fiemap.txt`.

[46] sysfs - the filesystem for exporting kernel objects. `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`.

[47] fio. `https://github.com/axboe/fio`.

[48] filebench. `https://github.com/filebench/filebench/wiki`.

[49] Stream ids and i/o hints. `https://lwn.net/Articles/685499/`.

[50] Redis. `https://redis.io`.

[51] Redis persistence. `https://redis.io/topics/persistence`.

[52] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[53] tc(8). `https://linux.die.net/man/8/tc`.

[54] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving ext4 for shingled disks. In *Proceedings of FAST'17: 15th USENIX Conference on File and Storage Technologies*, volume 1, page 105, 2017.

[55] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Anvil: Advanced virtualization for modern non-volatile memory devices. In *FAST*, pages 111–118, 2015.

[56] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *FAST*, pages 45–58, 2013.

[57] Sangwook Kim, Hwanju Kim, Sang-Hoon Kim, Joonwon Lee, and Jinkyu Jeong. Request-oriented durable write caching for application performance. In *USENIX Annual Technical Conference*, pages 193–206, 2015.

[58] Xiongzi Ge, Zhichao Cao, Pradeep Ganesan, David Du, and Dennis Hahn. Onestore: Integrating local and cloud storage with access hints. In *SoCC poster session*, 2014.

[59] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proceedings of the VLDB Endowment*, 5(10):1076–1087, 2012.

[60] David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher, Yongdae Kim, Yingping Lu, Aravindan Raghuveer, and Sarah Sharafkandi. Experiences in building an object-based storage system based on the osd t-10 standard. In *Proceedings of MSST*, volume 6, 2006.

[61] A storage standards update. `https://lwn.net/Articles/684264/`.

[62] Dingshan He, Xianbo Zhang, David HC Du, and Gary Grider. Coordinating parallel hierarchical storage management in object-base cluster file systems. In *Proceedings of the 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2006.

[63] Tracy F Sienknecht, Richard J Friedrich, Joseph J Martinka, and Peter M Friedenbach. The implications of distributed data in a commercial environment on the

design of hierarchical storage management. *Performance Evaluation*, 20(1-3):3–25, 1994.

[64] Application tiering and tiered storage. https://partnerdirect.dell.com/sites/channel/en-us/documents/cb120_application_tiering_and_tiered_storage.pdf.

[65] Fully automated storage tiering (fast). https://www.emc.com/corporate/glossary/fully-automated-storage-tiering.html.

[66] Ibm ds8880 hybrid storage. https://www-03.ibm.com/systems/storage/hybrid-storage/ds8000/index.html.

[67] Adaptive optimization for hpe 3par storeserv storage. https://www.hpe.com/h20195/V2/GetPDF.aspx/4AA4-0867ENW.pdf.

[68] Netapp virtual storage tier. http://www.netapp.com/us/media/ds-3177-0412.pdf.

[69] Cheng Li, Philip Shilane, Fred Douglis, Darren Sawyer, and Hyong Shim. Assert (! defined (sequential i/o)). In *HotStorage*, 2014.

[70] The disksim simulation environment (v4.0). http://www.pdl.cmu.edu/DiskSim/.

[71] Msr cambridge traces. http://iotta.snia.org/traces/388.

[72] Netapp e-series product comparison. http://www.netapp.com/us/products/storage-systems/e5400/e5400-product-comparison.aspx.

[73] Alberto Miranda and Toni Cortes. Craid: online raid upgrades using dynamic hot data reorganization. In *FAST*, pages 133–146, 2014.

[74] Ishwar Krishnan Sethi and GPR Sarvarayudu. Hierarchical classifier design using mutual information. *IEEE Transactions on pattern analysis and machine intelligence*, (4):441–445, 1982.

[75] Xin Li and Dan Roth. Learning question classifiers. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics, 2002.

[76] Michael P Mesnier, Matthew Wachs, Raja R Sambasivan, Alice X Zheng, and Gregory R Ganger. Modeling the relative fitness of storage. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 37–48. ACM, 2007.

[77] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. Automated lookahead data migration in ssd-enabled multi-tiered storage systems. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[78] Kernel asynchronous i/o (aio) support for linux. `http://lse.sourceforge.net/io/aio.html`.

[79] Bumjoon Seo, Sooyong Kang, Jongmoo Choi, Jaehyuk Cha, Youjip Won, and Sungroh Yoon. Io workload characterization revisited: A data-mining approach. *IEEE Transactions on Computers*, 63(12):3026–3038, 2014.

[80] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 43–56. ACM, 2011.

[81] Vasily Tarasov, Santhosh Kumar, Jack Ma, Dean Hildebrand, Anna Povzner, Geoff Kuenning, and Erez Zadok. Extracting flexible, replayable models from large block traces. In *FAST*, volume 12, page 22, 2012.

[82] Christina Delimitrou, Sriram Sankar, Aman Kansal, and Christos Kozyrakis. Echo: Recreating network traffic maps for datacenters with tens of thousands of servers. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 14–24. IEEE, 2012.

[83] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *FAST*, pages 59–72, 2013.

[84] Windows superfetch. `https://msdn.microsoft.com/en-us/library/windows/hardware/dn653317(v=vs.85).aspx`.

[85] Lipeng Wan, Zheng Lu, Qing Cao, Feiyi Wang, Sarp Oral, and Bradley Settlemyer. Ssd-optimized workload placement with adaptive learning and classification in hpc environments. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–6. IEEE, 2014.

[86] Zhengyu Yang, Jianzhe Tai, and Ningfang Mi. Grem: Dynamic ssd resource allocation in virtualized storage systems with heterogeneous vms. In *35th IEEE International Performance Computing and Communications Conference (IPCCC). IEEE*, 2016.

[87] Juniper wx/wxc series. `http://www.juniper.net/support/eol/wx_hw.html`.

[88] Riverbed wan optimization. `http:http://www.riverbed.com/products/steelhead/`.

[89] Cisco wae-7371. `http://www.cisco.com/c/en/us/td/docs/app_ntwk_services/waas/wae/installation/7341-7371/guide/7341gd/7300spec.html`.

[90] Infineta dms. `http://www.infostor.com/imagesvr_ce/4166/ESG%20Lab%20Validation%20Infineta%20Data%20Mobility%20Switch%20June%2011.pdf`.

[91] Emir Halepovic, Carey Williamson, and Majid Ghaderi. Enhancing redundant network traffic elimination. *Computer Networks*, 56(2):795–809, 2012.

[92] Haoyu Song and John W Lockwood. Efficient packet classification for network intrusion detection using fpga. In *Proceedings of the FPGA'05*, pages 238–245. ACM, 2005.

[93] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. Netfpga: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7. ACM, 2008.

[94] Muhammad Bilal Anwer and Nick Feamster. Building a fast, virtualized data plane with programmable hardware. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 1–8. ACM, 2009.

[95] Ashok Anand, Vyas Sekar, and Aditya Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 87–98. ACM, 2009.

[96] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of the SYSTOR 2009*, SYSTOR '09, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[97] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[98] Dong Yin, Deepak Unnikrishnan, Yong Liao, Lixin Gao, and Russell Tessier. Customizing virtual networks with partial fpga reconfiguration. *ACM SIGCOMM Computer Communication Review*, 41(1):125–132, 2011.

[99] Austin Appleby. Murmurhash 64 bits variant. `https://sites.google.com/site/murmurhash/`.

[100] Michael O Rabin. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[101] Yi Liu, Xiongzi Ge, David Hung-Chang Du, and Xiaoxia Huang. Par-bf: a parallel partitioned bloom filter for dynamic data sets. *The International Journal of High Performance Computing Applications*, 30(3):259–275, 2016.

[102] De5-net fpga development kit. `http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=526`.

[103] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceeding of the IEEE INFOCOM'99*, volume 1, pages 126–134. IEEE, 1999.

[104] Aaron Gember, Robert Grandl, Junaid Khalid, and Aditya Akella. Design and implementation of a framework for software-defined middlebox networking. In *Proceedings of the ACM SIGCOMM 2013*, pages 467–468. ACM, 2013.

[105] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th NSDI*, pages 459–473. USENIX, 2014.

[106] Dong Yin, Deepak Unnikrishnan, Yong Liao, Lixin Gao, and Russell Tessier. Customizing virtual networks with partial fpga reconfiguration. *ACM SIGCOMM Computer Communication Review*, 41(1):125–132, 2011.

[107] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Proc. 11th USENIX NSDI*, pages 543–546. USENIX Association, 2014.

[108] Enhanced platform awareness for pcie devices. `https://wiki.openstack.org/wiki/Enhanced-platform-awareness-pcie`.

[109] Sean C Rhea, Kevin Liang, and Eric Brewer. Value-based web caching. In *Proceedings of the 12th international conference on World Wide Web*, pages 619–628. ACM, 2003.

[110] Dingde Jiang, Lei Shi, Peng Zhang, and Xiongzi Ge. Qos constraints-based energy-efficient model in cloud computing networks for multimedia clinical issues. *Multimedia Tools and Applications*, pages 1–22, 2016.

[111] Dingde Jiang, Zhengzheng Xu Xu, Jindi Liu, and Wenhui Zhao. An optimization-based robust routing algorithm to energy-efficient networks for cloud computing. *Telecommunication Systems*, pages 1–10, 2015.

[112] Eyal Zohar, Israel Cidon, and Osnat Ossi Mokryn. The power of prediction: Cloud bandwidth and cost reduction. In *Proceedings of the SIGCOMM 2011*, volume 41, pages 86–97. ACM, 2011.

[113] Timothy Wood, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. Cloudnet: dynamic pooling of cloud resources by live wan migration of virtual machines. In *ACM SIGPLAN Notices*, volume 46, pages 121–132. ACM, 2011.

[114] Phlip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (TOS)*, 8(4):13, 2012.

[115] Sejun Song, Daehee Kim, Hyungbae Park, Baek-Young Choi, and Taesang Choi. Co-reduce: Collaborative redundancy reduction service in software-defined networks. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 61–66. ACM, 2015.

[116] Deepak Unnikrishnan, Ramakrishna Vadlamani, Yong Liao, Abhishek Dwaraki, Jérémie Crenne, Lixin Gao, and Russell Tessier. Scalable network virtualization using fpgas. In *Proceedings of the FPGA'10*, pages 219–228. ACM, 2010.

[117] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *ACM SIGCOMM Computer Communication Review*, 41(4):183–194, 2011.

[118] NM Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[119] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[120] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI*, pages 323–336, 2012.

[121] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[122] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 27–38. ACM, 2013.

[123] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[124] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on NSDI*, pages 459–473, Seattle, WA, April 2014. USENIX Association.

Appendix A

Glossary and Acronyms

A.1 Acronyms

Table A.1: Acronyms

Acronym	Meaning
SCM	Storage Class Memory
SSD	Solid State disk
SMR	Shingle Magnetic Recording
NRE	Network Redundancy Elimination
CDC	Content-Defined Chunking
VNRE	Virtualized NRE
RE	Redundancy Elimination
HDD	Hard Disk Drive
PMR	Perpendicular Magnetic Recording
PCM	Phase Change Memory
IOPS	I/Os per second
SCSI	Small Computer Systems Interface
LBA	Logical Block Address
HTTP	Hyper Text Transfer Protocol

Continued on next page

Table A.1 – continued from previous page

Acronym	Meaning
HTTPS	HTTP over Transport Layer Security
DRAM	Dynamic Random Access Memory
DM	Device Mapper
LVM	Logical Volume Manager
API	Application Program Interface
POSIX	Portable Operating System Interface
HSM	Hierarchical Storage Management
SAN	Storage Area Network
VM	Virtual Machine
FPGA	Field-Programmable Gate Arrays
TCP	Transmission Control Protocol
FP	Finger Print
WAN	Wide Area Network
WOA	Wide Area Network Optimization Accelerators
PR	Partially Reconfigurable
MB	Middle Box
AVI	Audio Video Interleaved
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
DMA	Direct Memory Access
SMTP	Simple Mail Transfer Protocol
FTP	File Transfer Protocol
RTSP	Real Time Streaming Protocol
DPI	Deep Packet Inspection
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
IaaS	Infrastructure as a Service
TLP	Transaction Layer Packet
DPD	Deduplication-Pending Data

Continued on next page

Table A.1 – continued from previous page

Acronym	Meaning
DPDK	Data Plane Development Kit
DFA	Deterministic Finite Automata