HintStor: A Framework to Study I/O Hints in Heterogeneous Storage

XIONGZI GE, NetApp, USA
ZHICHAO CAO and DAVID H. C. DU, University of Minnesota, Twin Cities, USA
PRADEEP GANESAN and DENNIS HAHN, NetApp, USA

To bridge the giant semantic gap between applications and modern storage systems, passing a piece of tiny and useful information, called I/O access hints, from upper layers to the storage layer may greatly improve application performance and ease data management in storage systems. This is especially true for heterogeneous storage systems that consist of multiple types of storage devices. Since ingesting external access hints will likely involve laborious modifications of legacy I/O stacks, it is very hard to evaluate the effect and take advantages of access hints. In this article, we design a generic and flexible framework, called HintStor, to quickly play with a set of I/O access hints and evaluate their impacts on heterogeneous storage systems. HintStor provides a new application/user-level interface, a file system plugin, and performs data management with a generic block storage data manager. We demonstrate the flexibility of HintStor by evaluating four types of access hints: file system data classification, stream ID, cloud prefetch, and I/O task scheduling on a Linux platform. The results show that HintStor can execute and evaluate various I/O access hints under different scenarios with minor modifications to the kernel and applications.

CCS Concepts: • **Information systems → Hierarchical storage management;**

Additional Key Words and Phrases: I/O access hints, heterogeneous storage systems, block storage, data management

ACM Reference format:
Xiongzi Ge, Zhichao Cao, David H. C. Du, Pradeep Ganesan, and Dennis Hahn. 2022. HintStor: A Framework to Study I/O Hints in Heterogeneous Storage. *ACM Trans. Storage* 18, 2, Article 18 (March 2022), 24 pages.
https://doi.org/10.1145/3489143

1 INTRODUCTION

Conventional **hard disk drives (HDDs)** have dominated the storage world for more than half a century. The mechanical parts, especially the read-write heads working with the rotating platters, usually lead to slow random I/O performance. In recent years, some variants of HDDs have come out, such as **Perpendicular Magnetic Recording (PMR)** [58], **Shingle Magnetic Recording**

This work was partially supported by NSF I/UCRC Center for Research in Intelligent Storage with NetApp collaboration and the following NSF awards 1439662 and 1812537.
Authors' addresses: X. Ge, NetApp, 7301 Kit Creek Road, Research Triangle Park, NC, 27709, USA; email: xiongzi@netapp.com; Z. Cao, University of Minnesota, Twin Cities, 4-192 Keller Hall, 200 Union Street SE, Minneapolis, MN, 55455, USA; email: caoxx380@umn.edu; D. H. C. Du, University of Minnesota, Twin Cities, 4-192 Keller Hall, 200 Union Street SE, Minneapolis, MN, 55455, USA; email: du@umn.edu; P. Ganesan, 7886 169 Street, Surrey, BC, V4N0J4, Canada Dennis Hahn, 13111 Castlewood Cir, Wichita, KS 67230 USA; email: gp.esgan@gmail.com.

1553-3077/2022/03-ART18 $15.00
https://doi.org/10.1145/3489143

(SMR) [46, 47, 70, 73], and **Interlaced Magnetic Recording (IMR)** [24, 72, 74] drives, which have more than 10 TB capacity for a single disk. These HDD variants still rely on data recording on magnetic media, though. Thus, disk seeking overhead remains high.

Over the past two decades, several new storage technologies/devices emerged and became mature besides the new HDD technologies. The high-performance storage devices, such as flash-based **Solid-State Drives (SSDs)** [21] and **Storage-level Class Memory (SCM)** [52] make storage faster. SSDs and SCM fundamentally differ from traditional HDDs. They can read/write data randomly without moving any mechanical arm or heads. In addition, flash-based SSDs have to erase a data block before serving new incoming write requests to the block. Thus, SSDs have asymmetric read/write speeds. SCM like **Phase Change Memory (PCM)** and STT-RAM are referred to high-speed and non-volatile storage mediums. Although SCM has an impressive performance (hundreds of thousands IO per second), its cost is still expensive [52]. A new service model called cloud storage, which stores data on remote systems, became popular recently. This usually involves more network latency when accessing the data outside of the cloud. Taking into consideration performance, capacity, and cost, storage systems are built in a more composite way, incorporating the emerging storage technologies/devices, including SCM, SSD, HDD, and even remote cloud storage. We consider this type of storage system heterogeneous.

Modern storage systems have mainly evolved from the old hard disk drives, such as the **Small Computer Systems Interface (SCSI)** architecture[63], which defines the data protocol to access a set of peripheral devices with a few parallel I/O buses. Such SCSI command set is still used widely to connect various buses (e.g., SATA, iSCSI) in Linux operating systems [31]. From the recent Linux storage stack diagram [64], multiple independent layers are developed and exist in the storage stack in a hierarchical manner. LVM based on **Device Mapper (DM)** [54] can manage multiple storage devices and provide a series of homogenous **logical block addresses (LBAs)** to the upper-level layers like filesystems. Such a simple and stable interface make the upper layers such as filesystems and applications evolve in an independent way [49]. For example, a file system is always built on a virtualized logical device that can be treated as a list of block addresses. Nevertheless, such a logical device may consist of several different types of storage devices.

To balance the storage capacity and performance, in a tiered-storage system, 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 typically based on the statistics of data access frequency. If the storage manager moves cold data too conservatively, then the I/O latency may suffer when there is not sufficient space for hot data to be migrated/stored to the fast-tier. However, an aggressive data migration policy may lead to unnecessary data transfers 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 [16]. Besides, I/O latency may be affected during the I/O path, as multiple components exist [67]. In addition, the virtualization layers may cause it to be more challenging to efficiently allocate storage resources [68].

Although extracting file system semantics [23] can help aggregate the correlated files into groups, the gap between applications and block storage still impedes efficient storage management between storage devices. The block layer that manages the storage space lacks the necessary information about file data, such as file metadata and data blocks belonging to the file. This makes block layer hard to appropriately allocate resources and orchestrate data blocks across different storage tiers based on their access patterns. In addition, applications from user-level or system-level may have different priorities. For example, some maintenance tasks (e.g., rsync), designed to improve data availability or system performance [3], may need to traverse the whole directory hierarchy and thus should be treated as background tasks. These I/O requests, however, may be

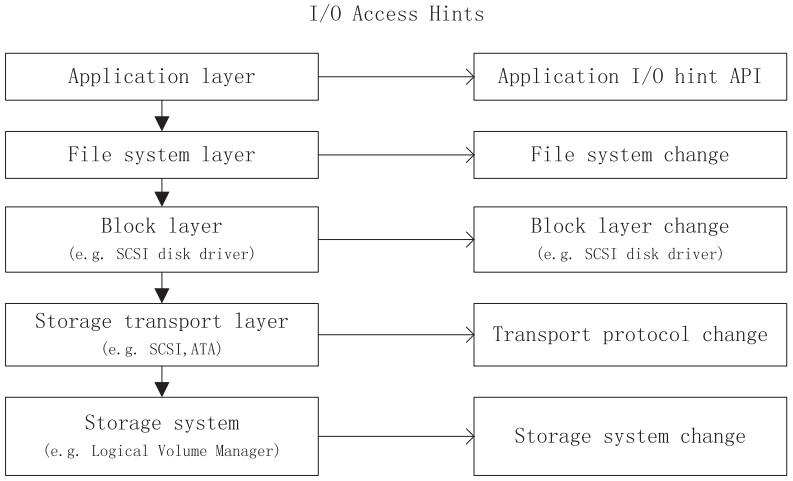

Fig. 1. Multiple layers of change when adding access hints to the current I/O stack.

mixed with some critical foreground I/O requests. Storage systems should prioritize the latter ones if they can. The backup data blocks are expected to stay on low-speed higher capacity storage devices. However, the lower-layer storage manager usually is not able to fully distinguish the lower priority I/O requests from the higher priority I/O requests as well as in which tier should the data stay.

To narrow the semantic gap between storage systems and their upper layers, one of the promising approaches is using I/O access hints [48, 49]. 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, then the storage manager 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 Reference [49] that allowed the controller in the back-end storage system 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 et al. [48] 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.

One of the major challenges of studying access hints in heterogeneous storage systems is lacking a common platform for evaluation. To send hints to the storage layer, different applications are required to have an efficient way of communicating with the storage layer (e.g., sysfs, SCSI commands). Figure 1 shows that multiple layers may need to be modified in the current I/O stack to implement and evaluate I/O hints. Developing hints and evaluating their effectiveness involves changing the applications, file systems, block drivers, transport protocols, and underlying storage subsystems. For example, in an SCSI subsystem, the hints info from the upper level may be written to a part of the CDB in an SCSI command. Not only the drivers in the block layer, but also the SCSI initiators in the transport layer may be involved in the implementation. This results in tedious work before a particular access hint can be evaluated for its effectiveness.

One of the industrial efforts has been conducted to standardize the T10 or T13 SCSI-based I/O hints [61]. This standardization work brought a lot of interest from different vendors by SNIA;

to the best of our knowledge, however, the proposed SCSI commands have not been fully implemented by any of the disk vendors or storage companies. There are probably two major reasons: one is that implementing the new commands requires the agreement across different layers (e.g., disk drivers, transport protocols) so the whole system can benefit from the change. The other reason is likely from the rapid development of NVMe drives [69], which can take advantage of the fast speed of flash by using more parallel data paths.

In this article, we design a generic and flexible framework, called HintStor, to study access hints in heterogeneous storage systems. The major goals are to devise an efficient framework for quickly evaluating various access hints and demonstrate a proof-a-concept that HintStor may be implemented in a production system in a simple way. To accomplish this goal, we design and implement a new application/user-level interface, a file system plugin, and a block storage data manager in HintStor. The purpose of HintStor is not to investigate efficient delivery mechanisms for each access hint, since such mechanisms remain to be defined and agreed by relevant standard organizations such as SCSI T10 and T13. We implement a prototype of HintStor in Linux. All implemented interfaces only require few modifications to file systems and applications. For the baseline storage management, HintStor triggers data migration by the block-level statistics (e.g., heatmap) and handles I/O in a FIFO manner. With HintStor, the users can quickly play with access hints and evaluate their efficiency and effectiveness on storage-level data management. Specifically, HintStor has the following key designs to simplify access hints implementation and evaluation:

- A new application/user-level interface allowing users to define and configure new access hints.
- A file system plugin in the VFS level extracting file layout information and defining a file-level data classification library for common file systems.
- A block storage data manager implementing four atomic access hint operations and triggering data movement and I/O scheduling in Linux based on DM. As a result, different policies associated with upper-level hints can be executed and evaluated.

To show the flexibility of the proposed framework, we evaluate different configurations with various types of storage devices (e.g., SSD, HDD, SCM, and cloud-based storage). Specifically, we implement and evaluate the following access hints to demonstrate the flexibility, simplicity, and effectiveness of HintStor:

- File system internal data classification. We use internal file system information to classify the data via different dimensions such as metadata/data, file size, and so on.
- Stream ID. The stream ID is used to classify different data and have the associated ones stored together or closely located on one device.
- Cloud prefetch. In this case, we study how access hints can help efficiently integrate on-premise storage and cloud storage.
- I/O task scheduling. The users can send hints from applications to the block storage to differentiate foreground tasks and background tasks.

The remaining of this article is organized as follows: In Section 2, we present the background of our work. Section 3 describes the design of HintStor. In Section 4, we demonstrate four different access hints with several storage configurations and show the evaluation results. Section 5 summarizes the related work. We conclude this article in Section 6.

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 [8, 48], caching

and prefetching in the host machine are mainly considered. Some off-the-shelf system calls in Linux, such as *posix_fadvise*() [40] and *ionice*() [41], may be used for access hints. For example, *sys_fadvise*64_64() [40] can specify the random access flag to the kernel so the kernel can choose appropriate read-ahead and caching techniques to improve access speed to the corresponding file.

However, this flag in *sys_fadvise*64_64() [40] is used in the Linux kernel and performed in the page cache level. Page cache usually refers to storing data in main memory. While in enterprise storage systems, storage itself may contain different types of storage devices like flash and SCM, which can be served as portions 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. Nonetheless, the existing prefetching engine in Linux does not support dynamic data movement in such a storage configuration. Thus, *fadvise*() and *ionice*() can help improve I/O performance via readahead mechanism but not sufficient to trigger efficient data movement in block level.

MPI-IO hints are used to optimize file systems in **high performance computing (HPC)** environment, such as Lustre [53] and GPFS [44]. Although some of the hints (e.g., *access_style* indicating the access pattern of the file [51]) are similar as the hints tested in HintStor, they need the support in the underlying storage to trigger data movement for HPC applications. GPFS also supports some general I/O hints [44], like not closing a file too frequently. XFS supports hints to do direct I/O optimization [4]. Similar to the readahead mechanism, these works also mainly focus on improving the buffer/cache layer in storage systems. Red Hat has enabled I/O limits in user space to help doing large block alignment (e.g., 4 KB) for devices from different vendors [56]. However, it lacks the support of data movement across multiple layers.

Some existing file systems support customized classifications. For example, *btrfs* can support different volumes in the same file system [26]. 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 implement I/O hints, without standard APIs, we may need to modify each application and even every layer. The current SCSI T-10 and T13 organizations have proposed some initial standardized interfaces for access hints [62]. However, due to a large portion of the modification of the I/O stack and applications, there is still no public implementation of such new standards. In addition, the widely used NVMe protocol reduces the interest of SCSI-based implementation. Our work is trying to build a flexible framework for both legacy and modern computing systems to design and evaluate the access hints from different layers. By means of slight modifications of each layer, we built HintStor to effectively provide access hints before a set of delivery mechanisms of access hints can be agreed by standard organizations.

3 HINTSTOR

3.1 Overview of the 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 structure, called *bio* [10]. The core of *bio*, *bi_io_vec*, links multiple fixed-size pages (4 KB). To manage and implement I/O access hints, we need to manipulate data in the block storage layer. The modularized kernel allows inserting new device drivers. **Device Mapper (DM)** [54] is an open-source framework in almost all Linux systems to provide a composited volume with differentiated block storage using various mapping policies (e.g., linear or mirror). Developers can build logical device drivers with various mapping

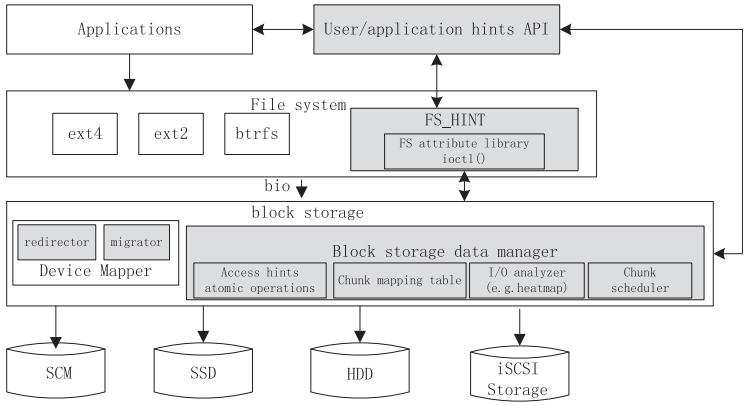

Fig. 2. HintStor framework.

policies in DM. Figure 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):

- 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/applications to send access hints to the storage layer.
- 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.
- To control data management of large size chunks, HintStor implements a new block storage manager in the kernel. It can implement policies associated with external access hints.

In the following sections, we will elaborate the design and implementation.

3.2 Prerequisite

We implement two generic block-level mapping mechanisms as two new target drivers in DM. These two targets are two basic functionalities that can be used in HintStor. They in total contain ~600 lines of C code in the kernel. They are named redirector and migrator and described below:

- **Redirector**: With the redirector, the target device ($bio \rightarrow bdev$) can be reset to the desired device (e.g., /dev/sdd). The control context of the redirector target inlcudes:

```
struct dm_dev * srcDev; /*Source device*/
struct dm_dev *dstDev; /*Destination device*/
struct metadata * mList; /*Mapping table */
unsigned long reads; /*Number of reads*/
unsigned long writes; /*Number of writes*/
```

  The mapping table (mList) 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 I/O requests associated with the *bio* data structure can be reassigned to the appointed device that belongs to the composited volume. An I/O request initially issued in source device, starting at LBA $start$ with offset, is abstracted as, $(srcDev, start, offset)$. HintStor can redirect it to the

destination device, with a new start LBA, $start1$, as ($dstDev$, $start1$, $offset$). The read count and write count are recorded periodically (e.g., every five minutes).

- **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 [28]. Data blocks in the migrator are grouped and divided into fixed-size chunks (a chunk contains a certain number of blocks). Each time the data blocks in a chunk associated with the I/O requests can be moved from one device to another device. Besides the above elements in Redirector, two more have been added to the control context of the Migrator target:

```
struct dm_kcopyd_client *copyClient;
/*The copy client for doing migration*/
bool isReplica; /*Replica flag*/
```

  As for copying a large number of blocks uses bandwidth, a throttle is set in the copyClient to limit the copy traffic. In addition, the migrator can be set up as a background process. HintStor also uses Migrator to carry out data replication functions. In the target driver interface, a flag isReplica is used to mark whether to keep or remove the original copy pointers after the migration.

The mapping table records the entry changes with each operation for both Redirector and Migrator. HintStor manipulates migration and redirection in chunks the size of which is much larger than a 512 B sector in DM. For a 100 GB volume with chunk size 1 MB, the total size of the mapping table is about 3 MB. The mapping table resides in the memory and is periodically written into the fastest tier (e.g., SSD or SCM). Inside both target drivers, read count and write count can help determine the destination when the composited volume has multiple components. Once the upper-level hints come down to the DM level, HintStor will make the decision based on the guidance from hints to copy the data or redirect the requests. As the goal of this study is not aiming to design specific storage policies, instead, we employ heuristic strategies and see how HintStor works.

We also modify the *dmsetup* tool in LVM2 [55] 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 (on 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 next.

3.3 Application/User-level Interface

Applications and users access data via interfaces such as file or object in the past decades. When an application opens a file, the OS will start to extract the file related information. For instance, although the user or the application usually is aware of the future accesses of the file (e.g., the access pattern), in the traditional Linux OS, they generally do not directly send such extra information to the back-end storage systems.

To accept and parse user-defined access hints, we want to simply use some scripts to set up attributes from user space to the kernel driver. HintStor makes use of the $sysfs$ interface [29] to build an attribute library for applications to communicate with block storage and file systems. $sysfs$ works as a pseudo-file system in Linux kernel that exports brief message from different kernel subsystems. We choose Reference [29], as it is simple and widely used by device drivers in Linux to dynamically configure parameters.

To make this interface work, we need to add the associated *sysfs* functions in the kernel to communicate with several new features in the block level, as shown in Figure 2. Then, the predefined commands from user space can be triggered and performed in the kernel. For example, if the user makes the migration request, then the command will be passed from the user space to the kernel. In the kernel, the migrator daemon will trigger chunk mapping retrieval process and finally block storage will be informed of which data chunks and where they are going to be migrated.

While there are different kinds of I/O hints, HintStor parses JSON files to ingest I/O access hints from users and applications. Once the scheme validation is passed, the extracted user-level access hints can be passed to the system. For some applications (e.g., cp, dd, rsync), HintStor can just simply configure the JSON interface file. For example, users can indicate task priority of *rsync* and then start the application with predefined scripts. There will be minor code changes of some applications such as filebench, though. In the current version, HintStor supports basic Linux operations such as *cp, dd, fsck, redis, rsync* key-value database, benchmark tools (e.g., fio [18] and filebench [17]). In addition, HintStor supports user-defined APIs such as stream ID and cloud prefetch. To demonstrate the scheduling assisted by upper-level hints, I/O priority numbers can be passed down associated with the files.

3.4 File System Plugin

After HintStor gets the upper-level I/O hints, it needs to find the location of the file data on a multi-tier storage system as the placement can affect the file access latency. Once the block level knows the data blocks associated with the upper-level requests, data migration can be triggered to speed up the future accesses. In the cases of foreground and background accesses, block-level can prioritize the former ones and later do the background I/Os. For both cases, the mapping from file location on the logical device can help the block-level storage understand the data distribution from each file. In addition, a filesystem usually consists of metadata blocks and data blocks. The metadata information includes block bitmap, inode, superblock, and so on. To improve filesystem performance, the metadata location is expected on a faster storage device, since this information is accessed more frequently.

File systems usually encapsulate the internal data management of files to provide a few POSIX APIs such as *read(), stat()* [11]. Each filesystem is implemented in its own way [11]. To classify filesystem internal data blocks and extract the file location information, HintStor provides a plugin in VFS level, called *FS_HINT*.

FS_HINT attempts to extract the filesystem metadata and its location. This includes file size, metadata/data locations. To date, *FS_HINT* supports the mainstream filesystems in current Linux, such as *ext2*, *ext4,* and *btrfs*. For example, *ext4* allocates a block for metadata via *ext4_new_meta_blocks*() according to the multi-block allocation mechanism, and *FS_HINT* captures the returned block group number (*ext4_fsblk_t*). The information collected is exposed to both user space and applications. This part is dedicated to each filesystem.

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. For the early support on file systems in HinStor, to simplify the implementation and meantime extract the necessary information to execute access hints, we only consider file systems that implement the*fiemap* callback on their *inode_operations* data structure. Most filesystems such as *ext4* and *btrfs* support extent. We use the *fiemap* ioctl() method from user space to retrieve the file extent mappings [27]. Without sector-to-sector mapping, *fiemap* returns the associated extents to a file. After the extraction is completed, the mapping information will be sent to the block level when certain policies can trigger data movement or I/O rescheduling. This process is transparent to the filesystem. The chunk-to-chunk mapping table is maintained in the block level, as discussed in Section 3.2.

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 is mainly used to study the data management at storage level with multiple storage devices, each of which has a large capacity compared with OS cache study. Thus, even if we sacrifice a little accuracy of a small portion of data blocks, the results will not be affected much. Consequently, *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 with most Linux platforms. We make the LBA to PBA mapping in a file as a function in *FS_HINT*. The major function in this part contains only about ∼50 LoC. Furthermore, *FS_HINT* supports querying the mappings for multiple files without complex kernel code modification.

3.5 Block Storage Data Manager

According to the above discussion, to trigger policies based on the access hints from upper levels, HintStor requires block-level storage operations. To make the underlying block layer perform access hints, we devise and implement a block storage data manager extending from DM using the two basic target drivers described in Section 3.2. It mainly contains a chunk mapping table, a chunk-level I/O analyzer, a chunk scheduler, and a model of access hint atomic operations.

The chunk mapping table maintains the **Logical Block Address (LBA)** to **Physical Block Address (PBA)** mapping. HintStor implements block-level data management with the granularity of fixed-size chunks. This simulates the scenario of enterprise storage management [22] where the chunk size can be in GB level. In addition, HintStor can configure different sizes of chunk size. The size of the mapping table, as discussed in Section 3.2, is small.

The chunk-level I/O analyzer is used to monitor 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 written in Perl is developed to visualize the real-time access statistics. HintStor internally has a periodical data migration policy that relies on the chunk access frequency to do data migration. This information might be inaccurate or improper for some cases to do access prediction; we can still use it as a metric to evaluate the correctness of data movement. For example, after migrating data chunks to the slower tier, the access frequency continuously increases, though. This may trigger the system to move or replicate the data into a faster layer.

The chunk scheduler is devised to evaluate the I/O scheduling according to priorities from different tasks. The block-level data manager implements a simple chunk-level I/O scheduler to reorder the accesses by dividing requests into different queues. In the current prototype, HintStor supports a work queue and a waiting queue. In default, HintStor uses FIFO to maintain the original access order, so every request goes to the head of the work queue. When the task priority hints arrive, the requests associated with the lower priority will be added to the end of the waiting queue. The

application can set up a timer to invoke the tasks and then the request will be moved from the wait queue to the work queue before the deadline.

HintStor treats data placement and data movement as essential functions in improving the performance of a heterogeneous storage system. 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 original requested I/O locations are redirected to another place. For example, in a composited logical volume with SSD and HDD, the data manager can issue multiple REDIRECT operations to send small files to SSD instead to HDD. When the *bio* request comes into DM, REDIRECT will reassign the data chunk to the destination location. The redirection function calls the redirector driver in DM and manages the redirected I/O requests.
- **MIGRATE:** Data migration plays a crucial role in HintStor. To enable data migration, a data migrator daemon is running in the background. When data blocks are originally placed in storage with differentiated storage devices, the data access frequencies may change from time to time. The MIGRATE operation moves the data chunks from the original place to the destination place via calling 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 the timer-based heuristic migration policy. For example, the user can configure the data migration every two hours 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 that is assigned by the applications. HintStor makes use of the migrator target driver in DM by slightly modifying the return process. This makes the mapping table keep the original copy pointer. Adding more replicas of the hot chunks across multiple devices will improve data availability and reduce the mean response time. We can use REPLICATE to create multiple duplications to emulate such cases. Like 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/Os to this chunk.

3.6 I/O Access Hint Classification

Basically, I/O access hints can be either statically extracted from the existing systems or dynamically captured in the systems and even added by the users/applications. Table 1 shows the two categories of access hints with 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, and so on. HintStor can decide the initial location of different types of data, according to the static access hints. To achieve better QoS for the applications, intelligent data movement plays an essential role to make use of the large capacity low-cost storage space (e.g., cloud storage or HDD) and avoid the longer network latency.

Table 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 migration interval chunk size
Dynamic I/O access hints	open a file write/read a file stream ID cloud prefetch task priority

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 that is opened again, such system calls will be captured to help block storage make prefetching decisions. 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. Access hints can be applied based on system level, process level, file level, and chunk level. HintStor provides the flexibility to execute different types of hints to evaluate the impact. The data management is based on the fixed size chunks the size of which is usually much larger than that of a typical page in memory management. Some of the previous work (e.g., Reference [49]) implemented I/O hints based on each I/O. HintStor handles I/O in chunk level and combines additional information, such as file structure and users' hints. Task scheduling in HintStor is managed by the process level. In Section 4, we will show how we use HintStor to carry out some of the access hints.

3.7 Baseline System vs. HintStor

Baseline system without access hints. Storage systems internally usually have their own data management to move data from time to time. HintStor provides the baseline framework to perform regular I/O requests. We build a tiered-based storage system using DM's framework. A heuristic and lightweight tiering mechanism is implemented in the block storage manager. The chunk scheduler is configured using the FIFO policy that does not affect the request order. The I/O analyzer collects the access statistics based on fixed-size data chunks. A heatmap was maintained internally for both visualization and querying. A fixed-size chunk data migration policy was implemented to move data between the faster-tier and the slower-tier periodically. Thus, basic data migration has already been enabled in the baseline system. Depending on the migration interval, the most frequently accessed chunks are usually placed on the fastest tier.

To reach the flexibility, HintStor supports modes to play hints under different configurations. The major purpose of this methodology is to evaluate broad use cases. For the baseline setup, only block-level heuristic periodic data migration is enabled. The user can configure the system parameters before running the evaluation. This includes:

- Baseline system migration interval. The migration interval can be used to evaluate how access hints work when the baseline system moves data frequently.

- Chunk size. When the chunk size is set to a smaller one, HintStor may achieve better accuracy but involves more cost in management.

3.8 Discussion

API support with changes in different layers. As discussed in Reference [60], changing any of the file systems to let them explicitly send hints to the underlying block storage systems has obvious benefits but also drawbacks. There is tradeoff of simplicity and the consensus from the standard change, though. For example, SCSI-based I/O hints have been proposed for years, while there is still a long way for the broad industry to make the agreement, and the industry is moving towards NVMe. HintStor does not implement the SCSI-level hints but making slight changes of the file system by plugging into an FS_HINT module into the VFS level. For some filesystems, if fiemap is not supported, then we can still use the block-level statistics such as heatmap and hints from the applications to assist data management. However, the accuracy of moving the data chunks may be reduced.

Production. While the initial goal of HintStor was to provide a simple and quick way to implement and evaluate access hints, HintStor has the potential to be developed as a product by some further efforts. For example, standardization of access hints should be made by the industry like what has been done on T10-based I/O hints for SCSI-based I/O subsystem. As a result, certain applications could be involved to make minor changes that can boost performance.

4 EVALUATION

This section will show how we perform and evaluate four types of I/O access hints:

- File system internal data classification. By extracting file system's internal structure, we can get the metadata and data information. In addition, HintStor uses a simple way to approximately find the mapping of the locations of each file. 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 helps storage systems to improve caching efficiency, space allocation, and system reliability (in flash memory-based storage devices), and so on. HintStor exposes an interface in user space so applications can assign a stream ID for writes in each application as well as across applications.
- Cloud prefetch. We mainly study how access hints can efficiently integrate on-premise storage and cloud storage. Data blocks are expected to automatically transfer from local servers to remote cloud storage.
- I/O task scheduling. Some tasks like maintenance tasks are treated as background tasks. HintStor designs a chunk scheduler that can parse I/O priority hints from the users. We will use rsync as an example in our experiments.

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.

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 faster devices and slower devices. To study file system access hints, we define a two-level file data classification model, as shown in Table 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

Table 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, ∞)

on a faster device. In our current implementation, HintStor does not distinguish different types of metdadata (e.g., inode info vs. journal).

For the cases of level-2-hints, the data manager not only places the blocks with file metadata information on the 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, then the data manager can select the file size smaller than a certain value (e.g., 1 MB) to store on SSD. The key is that HintStor enables the block storage layer to be aware of metadata as well as the size of a given file. We change the *sys_read*() system call to monitor file operations. HintStor takes action 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 *dm−linear* target driver [30] and there are not any access hints generated in the system. *FS_HINT* in HintStor is disabled in the no-hint cases.

We use filebench [17] to create a set of files. The file server contains 10,248 files in 20 directories. We create 48 files with 1 GB size and 200 files with 100 MB size. The sizes of the rest files are uniformly distributed among 1 KB, 4 KB, 6 KB, 64 KB, 256 KB, 1 MB, 4 MB, 10 MB, and 50 MB. The total capacity of the files is about 126 GB. We run two workloads. The first one is like the Fileserver workload in Filebench with read/write ratio 1:2. The other one is based on the webserver, where read/write ratio is about 10:1. Chunk size is set to 1 MB. The migration interval in the baseline system is set to every five minutes. We run each test for one hour so the data migration happens in the no-hint case. Each time filebench runs the same number of operations. In addition, each test is run three times and we show the average result. In the end of each test, Linux will trigger the flush() and sync() command to clear the dirty pages.

We first test a storage configuration with two different storage devices, HDD (120 GB) and SSD (40 GB). 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 10 MB to store on SSD. We compare the results with three mechanisms including no-hint, Level-1-hints, and Level-2-hints on ext4. As shown in Figure 3, the average I/O latency is reduced by adding file system data classification hints. Compared with the no-hint cases, by adding Level-1-hints, the latency on ext4 is reduced by 48.2% and 28.5% for the Fileserver and Webserver workloads, respectively. By further adding Level-2-hints, the latency is reduced by 12.1% and 18.9% compared with the cases only using Level-1-hints.

To demonstrate the flexibility with different types of devices and show the effect of different chunk sizes in HintStor, we build a heterogeneous storage with HDD (120 GB), SSD (35 GB), and SCM (5 GB). We use a fixed DRAM space (5 GB) to emulate an SCM device. We configured different values (1 MB, 4 MB, 8 MB, and 16 MB) of the chunk size during the initialization step. Based on Level-1-hints, the data manager will first place the data blocks on SCM. Based on Level-2-hints, files with sizes smaller than 256 KB will be placed on SCM. SSD is used to keep files with sizes from 256 KB to 100 MB. The files with sizes larger than 100 MB will be placed on HDD. If SCM runs out of space, then the data chunks will be evicted from SCM to SSD via LRU manner. When SSD is full, the same policy is used to evict data chunks from SSD to HDD. As shown in Figure 4, when the chunk size is 1 MB, by adding Level-1-hints, the average request latency is reduced by 40.1% compared with the no-hint cases. With Level-2-hints, the average I/O latency is reduced by 38.9%, compared with the Level-1-hint cases. When the chunk size is

Fig. 3. The average latency on a hybrid SSD and HDD volume.

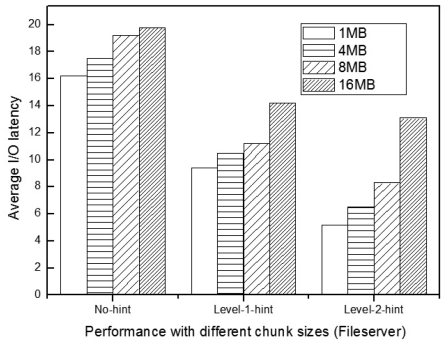

Fig. 4. Performance on a hybrid SCM, SSD, and HDD volume with different chunk sizes.

increased, there is less impact on the baseline system than in the HintStor in both Level-1-hints and Level-2-hints cases. The major reason is that HintStor moves data more frequently and replies more on fine-grained chunks. When the chunk size becomes larger, there are increasing chances of a chunk containing data from different patterns.

The previous studies 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.

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-1,000 frequently accessed chunks into the SCM in a fixed period. We can configure various migration intervals to investigate how data migration affects performance. HintStor calculates the accumulated statistics at the end of each interval and triggers data migration if it is needed.

Figure 5 and Figure 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 5 and Figure 6, the average I/O latencies in both ext4 and btrfs are reduced by adding access hints. Compared with the above cases, there is more improvement in this case where the storage system provides data migration management. Without hints, both performances can be improved by moving frequently accessed data into the fastest 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.

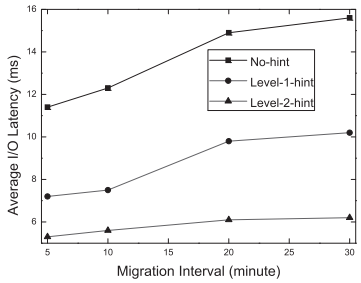

Fig. 5. The average latency with different migration intervals for ext4.

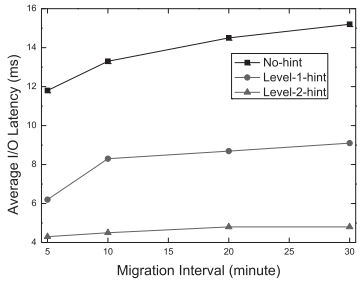

Fig. 6. The average latency with different migration intervals for btrfs.

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 calculate 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 1%. Thus, access hints operations are lightweight in HintStor.

From the experiments in Section 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. The results are consistent with prior work that put small files and metadata in the faster storage in Reference [49]. Besides, HintStor can evaluate access hints with various block-level data migration policies.

4.2 Stream ID

Employing application-level data classification usually requires modifications of each application. We take the idea of stream ID [14] as an example of the application-level access hints. In block storage, data requests may come from multiple sources (streams). Therefore, the data allocation and migration decisions based on analyzing the aggregated traffic may not be very effective. Stream ID is used to distinguish data sources and assist the storage to make proper data placement decisions.

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 the Linux kernel. If the system wants to distinguish the data based on a file system (one possible data source), 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

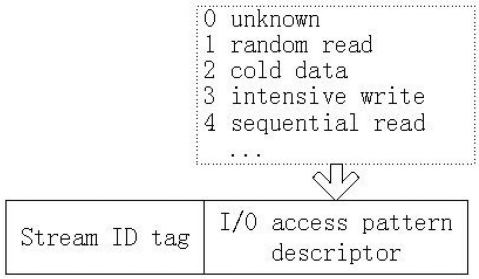

Fig. 7. Diagram of the enhanced stream ID.

all file systems. The applications just inform the devices the access patterns of the I/O requests so the devices can allocate space based on each stream.

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 such as file name and access patterns, they are implemented in the 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 enhanced stream ID hints including two fields, as shown in Figure 7. The second 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 leave it as default (unknown). The second field is the tag that records the stream ID number. We use the chunk mapping table in HintStor to record all the stream ID numbers and their patterns. The stream ID tag and pattern fileds currently take 1 byte in the reserved field of each mapping entry. We use both synthetic workload and a real key-value database to evaluate stream ID effectiveness.

We modified the *FB_open*() interface in filebench [17] 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 if it is not set by the user, the access pattern field is configured as "unknown." Note this 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. Please note that they could come from four filesytems, too. Each type of file is tested using one of the four predefined workloads: Webserver, Fileserver, Singlestreamwrite (iosize is set to 20 GB), and Randomwrite. In total, we create 1,560 files, with a total size of 125 GB. The hybrid volume is configured with 150 GB using 20 GB SSD, 125 GB HDD, and 5 GB 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 the storage pool free space.

Data blocks will be placed based on the stream ID hints when this information is set by the user. To leverage the properties of each device, the data manager will place the single stream write data on HDD, Webserver data on SSD, Randomwrite data on SCM. The data associated with the fileserver workload will be placed 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 Section 3. Random write pattern is prioritized to reside on SCM.

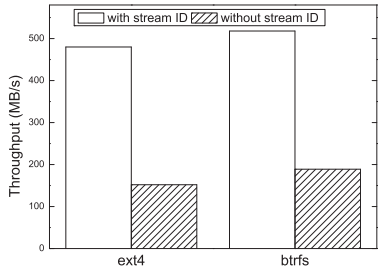

Fig. 8. System throughput improvement with stream ID.

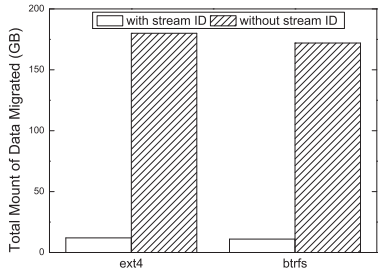

Fig. 9. The total amount of data migrated in HintStor with stream ID and without.

SSD is more appropriate for random read data. HDD is serving sequential patterns as well as cold data.

The migration timer is set to happen every 10 minutes. We run the test for two hours. We measure the system throughput with stream ID and without, respectively. As shown in Figure 8, system throughput in the case with stream ID outperforms the result of no stream ID case by 3.2× and 2.7× in ext4 and btrfs, respectively. Figure 9 shows the total amount of 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 *FB_open*(). Hence, there is little impact on the system performance by performing access hints in filebench.

4.3 Cloud Prefetch

Cloud storage emerges as mainstream for reducing cost and increasing data reliability. We can build storage services that are based on the integration of on-premise local storage and off-premise cloud storage. Such a 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 storage 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

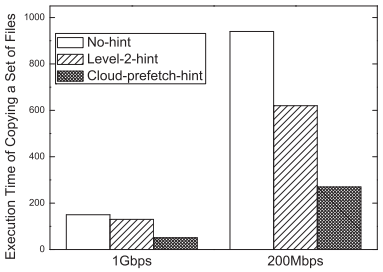

Fig. 10. Execution time of *cp*, a set of files that are placed across local and cloud storage.

user/application-level interface, applications can appoint which files to buffer either on the local side or on the 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 being used up.

In the experiment, we configure the local storage with 10 GB HDD and 1 GB SSD. HintStor currently does not support S3 interface. Although we do not use public cloud storage such as AWS EBS [2], Azure [50], and Google persistent disk [20], we can replace the iSCSI storage by attaching the cloud storage volume. 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 the cloud storage are configured as a single volume on top of which ext4 file system is mounted. We use the Linux traffic control tool *tc* [42] to mimic the network latency.

We use *dd* (generated from /dev/urandom) to create two 1 GB files and 100 1 MB 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 placed 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 a 1 GB local storage buffer for a Gbps connection and a 2 GB local storage buffer for a 200 Mbps connection. For the1 Gbps network, when we start to create the first 1 GB file, a cloud prefetch hint is sent to the block storage level. While for the 200 Mbps network, cloud prefetch is triggered along with the creation 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 Section 4.1), and cloud-prefetch-hint. Figure 10 shows the total execution time of each case. In the case of 1 Gbps 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 the 200 Mbps 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 the cloud storage in Figure 11. Assisted by the cloud prefetch, the file data is buffered on the local storage. As 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.

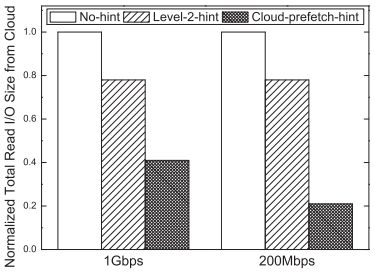

Fig. 11. Normalized total read I/O size from cloud storage.

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, then cloud prefetch hint will be disabled.

Concluding from this example, we can evaluate access hints in a storage system with on-premise and cloud storage in HintStor. We can configure the internal buffer space of the volume to play with different user-defined cloud prefetch hints.

4.4 I/O Task Scheduling

The above three examples are mainly about data movement impact. In the last experiment, we will study the I/O task scheduling along with access hints in HintStor. Though Linux has its own I/O scheduler, HintStor implements a separate chunk scheduler to fully maintain the scheduling in the block storage data manager so it can leverage the chunk mapping table and queue structure from the baseline storage system.

HintStor in default is configured as FIFO for each chunk access without any reordering so the block I/O scheduler can still work. The background task may access a large number of files and can affect the foreground applications significantly [3]. For all the foreground tasks, the incoming requests are added to the working queue. Requests associated with the lower priority tasks or background tasks are appended to the waiting queue. A timer along with the request can be set to enable invoking requests when the deadline reaches. When the working queue is empty, the block data manager will serve the wait queue. With HintStor's chunk scheduler, users can set up different priorities and evaluate the impact on the storage system.

In this experiment, we modify and run the rsync application as the background application. Rsync is used widely to do synchronization between the source and destination directories that can be on two different locations. It detects the differences between the two locations by the chunksum information and only does incremental file transfer. A process called sender in rsync scans the whole source directory. After sending the metadata to the destination, the destination will calculate the chucksum and send it back. If there are updated data blocks, then rsync starts to transfer them. We change a few lines of the sender program to add the priority access hints when it starts to access the source files. When there are other foreground tasks, the I/O requests to the rsync application will be in the waiting queue until the deadline arrives.

We use two servers to run the rsync application. HintStor only runs at the source. The destination directory is placed on the SSD layer to avoid high latency in the destination side. The workload generated by Filebench is treated as the foreground task.

A volume is created with 10 GB SSD and 50 GB HDD in the source. Then, we use Filebench to populate the volume with 50 GB files. The two servers are connected by a 10 Gbps local network. Then, we run the rsync application continuously to synchronize the two directories

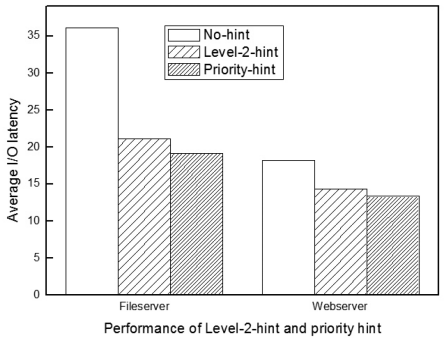

Fig. 12. Performance of HintStor with priority hints.

where filebench uses its subdirectory as the working directory. The chunk size is set to 1 MB. The migration interval is set to every five minutes in the baseline system. We test three cases (no-hint, Level-2-hint, and the priority-hint) using Fileserver and Webserver workloads.

As shown in Figure 12, Level-2-hint reduces the average I/O latency by 24.6% and 20.9% for Fileserver and Webserver workloads. By adding the priority hints, the latency number is further decreased by 5% and 6%.

This demonstrates that HintStor can play with basic I/O scheduling by informed hints from the user or application levels. With minor modification of the user space applications, the users can set up the priority information of the task and send it to the storage system. More scheduling algorithms can be added in the future.

5 RELATED WORK

As aforementioned, more new storage technologies are emerging and (e.g., SMR [1, 73], SCM [38, 52, 66]) making the storage systems more heterogeneous. Each storage media has its own idiosyncrasies with different costs and performances. To efficiently make use of hybrid storage devices in a storage system, the legacy way is building a tiered storage system [22, 25, 65, 71]. Such systems always make data tiering via monitoring I/O activities in block/chunk level or in the cloud [9]. Hermes [34] achieves the management for heterogeneous storage system with multiple layers of buffering. TDDFS can do tier-aware data deduplication in the file system level [7]. The limitation of legacy data tiering is the lack of sufficient information to distinguish block boundaries from different applications so as not being able to place relative data blocks together. Shettl et al. studied using the unsupervised machine learning technique to assist data migration in hybrid storage systems [59]. HintStor can implement such kind of work to incorporate more hints.

Researchers started to pay attention to introduce access hints into storage systems by enhancing caching and prefetching [8, 35, 48, 49]. Linux uses read-ahead system calls or related mechanisms to help the OS to adapt the conventional hard drivees to do conservative sequential prefetching. Aggressive prefetching is an effective technique for reducing the running time of disk-bound applications. Through speculatively pre-executing the code in an application, the OS may exploit some hints for its future read requests [8]. Even host-side cache is considered to improve write access to network-based storage [33]. SSD or fast storage devices can build hybrid cache along with DRAM where caching data can be placed across different storage devices [43]. To achieve the 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 [32]. As mentioned in the first section, Sonam Mandal et al. [48]

studied block-layer data deduplication by using hints. OneStore tries to integrate local and cloud storage with application-level access hints [15]. Strata [36] adopts the log-structured design to achieve the good tradeoff between performance, capacity, and feasibility at file system level. FIOS [6] extends the stream ID concept for multi-stream SSD to further reduce the write amplification. The workload hints are passed down to make better stream ID assignments.

Most available research regarding access hint typically emphasizes static access hints. They mainly study using hints extracted from the existing components, such as applications [75], file system [49], or database [45] to assist OS prefetching and caching (external caching disks) and even affect SSD firmware layer [75]. Intel recently released an I/O trace tool [19] that can capture metadata information as well as extended classification, but it does not allow for user-level hinting. Combining the trace analysis and replayer tools such as NetStorage [39], we might get more insights of the I/O in a whole system. Some researchers have studied static SCSI-level hints for SSD caching in Reference [49]. Intel Lab also released the new version of its Open Storage Toolkit [37], which can pass hints down to the underlying SCSI and NVMe layers. Besides the tracing ability, it contains a cache layer that can be used to evaluate hints along with caching policies. In Reference [45], they exploited how we could modify the existing system in databases 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, then this will help data blocks stay on the appropriate layers combining the tradeoff between cost and performance.

Du et al. implemented T10-based object-based storage [12], which tries to combine both block and file interfaces. Recently, there are standardization efforts in T10 [13, 62] and T13 [5] 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 applications could directly design and configure the stream ID for their writes [14]. Duet [3] explored page cache change hints to the background task to do less impact on the foreground task. This could improve the synergy of different tasks, but it does not consider the hybrid storage scenarios. Block liveness information was studied in Reference [60] to notify the underlying file systems of the status related to the blocks. Fstream [57] explores stream information in the file system for multi-stream SSDs.

To date, we have not seen an open-source framework that 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.

6 CONCLUSION

Using access hints on emerging heterogeneous storage systems to improve performance is critical. However, the implementation of access hints for system designers is very difficult. This article 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 using DM. With HintStor, storage systems composed of various storage devices can perform pre-devised data placement, space reallocation, data migration policies, and enhanced I/O scheduling 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 four types of access hints: file system data classification, stream ID, cloud prefetch, and I/O task scheduling in a Linux platform. The results demonstrate that HintStor can execute and evaluate various I/O access hints under different scenarios at a low cost of reshaping the kernel and applications. For the four access hint cases, we

have demonstrated the value of using access hints. However, our current designs are very simple. It is possible more elaborated designs of access hints can be considered in the future. The overhead of using HintStor is also obtained at about 1% of CPU usage and a small size memory for chunk mapping. Therefore, it is possible that HintStor can be used as a generic way to deliver access hints for certain applications. We plan to exploit this possibility in the future. We are planning to enrich the access hints library, enable more automatic hints extraction (e.g., machine learning), and support more platforms (e.g., container) and protocols (e.g., S3).

ACKNOWLEDGMENTS

We thank Al Andux, Jerry Fredin, Gaurav Makkar, David Slik, and Ranjeet Sudan from NetApp and all the members in the CRIS group for their suggestions and support. We also thank the anonymous reviewers for carefully reading our manuscript and providing many insightful comments and suggestions.

REFERENCES

[1] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. 2017. Evolving Ext4 for shingled disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*.
[2] Amazon. 2018. Amazon Elastic Block Store. Retrieved from https://aws.amazon.com/ebs/.
[3] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. 2015. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 457–473.
[4] ANL. 2018. Hints for XFS. Retrieved from http://ftp.mcs.anl.gov/pub/romio/users-guide/node7.html.
[5] Technical Committee T13 AT Attachment. 2017. Technical Committee T13 AT Attachment. Retrieved from http://www.t13.org/.
[6] Janki Bhimani, Ningfang Mi, Zhengyu Yang, Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2018. FIOS: Feature-based I/O stream identification for improving endurance of multi-stream SSDs. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'18). IEEE*.
[7] Zhichao Cao, Hao Wen, Xiongzi Ge, Jingwei Ma, Jim Diehl, and David H. C. Du. 2019. TDDFS: A tier-aware data deduplication-based file system. *ACM Trans. Stor.* 15, 1 (2019), 1–26.
[8] Fay Chang and Garth A. Gibson. 1999. Automatic I/O hint generation through speculative execution. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Vol. 99. 1–14.
[9] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. 2015. Cast: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 45–56.
[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005. *Linux Device Drivers: Where the Kernel Meets the Hardware*. O'Reilly Media, Inc.
[11] Daniel P. Bovet, and Cesati, Marco. 2007. Understanding the Linux kernel. (2007).
[12] David Du, Dingshan He, Changjin Hong, Jaehoon Jeong, Vishal Kher, Yongdae Kim, Yingping Lu, Aravindan Raghuveer, and Sarah Sharafkandi. 2006. Experiences in building an object-based storage system based on the OSD T-10 standard. In *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies*, Vol. 6.
[13] Jake Edge. 2016. A storage standards update. Retrieved from https://lwn.net/Articles/684264/.
[14] Jake Edge. 2016. Stream IDs and I/O hints. Retrieved from https://lwn.net/Articles/685499/.
[15] Xiongzi Ge, Zhichao Cao, Pradeep Ganesan, David Du, and Dennis Hahn. 2014. OneStore: Integrating local and cloud storage with access hints. In *Proceedings of the Symposium on Cloud Computing Poster Session*.
[16] Xiongzi Ge, Xuchao Xie, David H. C. Du, Pradeep Ganesan, and Dennis Hahn. 2018. ChewAnalyzer: Workload-aware data management across differentiated storage pools. In *Proceedings of the IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 94–101.
[17] Github. 2017. filebench. Retrieved from https://github.com/filebench/filebench/wiki.
[18] Github. 2017. fio. Retrieved from https://github.com/axboe/fio.
[19] Github. 2021. Standalone Linux IO Tracer. Retrieved from https://github.com/Open-CAS/standalone-linux-io-tracer.
[20] Google. 2018. Persistent Disk. Retrieved from https://cloud.google.com/persistent-disk/.
[21] Jim Gray and Bob Fitzgerald. 2008. Flash disk opportunity for server applications. *Queue* 6, 4 (2008), 18–23.
[22] Jorge Guerra, Himabindu Pucha, Joseph S. Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost effective storage using extent-based dynamic tiering. In *Proceedings of the Conference on File and Storage Technologies*, Vol. 11. 20–20.

[23] Yu Hua, Hong Jiang, Yifeng Zhu, Dan Feng, and Lei Tian. 2009. SmartStore: A new metadata organization paradigm with semantic-awareness for next-generation file systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 10.
[24] Euiseok Hwang, Jongseung Park, Richard Rauschmayer, and Bruce Wilson. 2016. Interlaced magnetic recording. *IEEE Trans. Magnet.* 53, 4 (2016), 1–7.
[25] Elena Kakoulli, Nikolaos D. Karmiris, and Herodotos Herodotou. 2018. OctopusFS in action: Tiered storage management for data intensive computing. *Proc. VLDB Endow.* 11, 12 (2018), 1914–1917.
[26] Kernel.org. 2017. btrfs. Retrieved from https://btrfs.wiki.kernel.org.
[27] Kernel.org. 2017. Fiemap ioctl. Retrieved from https://www.kernel.org/doc/Documentation/filesystems/fiemap.txt.
[28] Kernel.org. 2017. kcopyd. Retrieved from https://www.kernel.org/doc/Documentation/device-mapper/kcopyd.txt.
[29] Kernel.org. 2017. sysfs - The filesystem for exporting kernel objects. Retrieved from https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.
[30] kernel.org. 2019. dm-linear. Retrieved from https://www.kernel.org/doc/Documentation/device-mapper/linear.txt.
[31] Kernel.org. 2021. SCSI Interfaces Guide. Retrieved from https://www.kernel.org/doc/html/v4.13/driver-api/scsi.html.
[32] Sangwook Kim, Hwanju Kim, Sang-Hoon Kim, Joonwon Lee, and Jinkyu Jeong. 2015. Request-oriented durable write caching for application performance. In *Proceedings of the USENIX Annual Technical Conference*. 193–206.
[33] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. 2013. Write policies for host-side flash caches. In *Proceedings of the Conference on File and Storage Technologies*. 45–58.
[34] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: A heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-performance Parallel and Distributed Computing*. ACM, 219–230.
[35] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An informed storage cache for deep learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 283–296.
[36] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 460–477.
[37] Intel Labs. 2021. Open Storage Toolkit from Intel Labs. Retrieved from https://sourceforge.net/projects/intel-iscsi/.
[38] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 257–270.
[39] Bingzhe Li, Hao Wen, Farnaz Toussi, Clark Anderson, Bernard A. King-Smith, David J. Lilja, and David H. C. Du. 2019. NetStorage: A synchronized trace-driven replayer for network-storage system evaluation. *Perform. Eval.* 130 (2019), 86–100.
[40] Linux. 2017. fadvise(2). Retrieved from https://linux.die.net/man/2/fadvise.
[41] Linux. 2017. ionice(1). Retrieved from https://linux.die.net/man/1/ionice.
[42] Linux. 2017. tc(8). Retrieved from https://linux.die.net/man/8/tc.
[43] Yi Liu, Xiongzi Ge, Xiaoxia Huang, and David H. C. Du. 2015. MOLAR: A cost-efficient, high-performance SSD-Based hybrid storage cache. *Comput. J.* 58, 9 (2015), 2061–2078.
[44] LRZ. 2018. Using GPFS. Retrieved from https://www.lrz.de/services/compute/supermuc/filesystems/gpfs-usage/.
[45] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. 2012. hStorage-DB: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.* 5, 10 (2012), 1076–1087.
[46] Peter Macko, Xiongzi Ge, John Haskins Jr, James Kelley, David Slik, Keith A. Smith, and Maxim G. Smith. 2017. SMORE: A cold data object store for SMR drives (extended version). *arXiv preprint arXiv:1705.09701* (2017).
[47] Peter Macko, Xiongzi Ge, J. Kelley, D. Slik, et al. 2017. SMORE: A cold data object store for SMR drives. In *Proceedings of the 34th Symposium on Mass Storage Systems and Technologies*.
[48] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. 2016. Using hints to improve inline block-layer deduplication. In *Proceedings of the Conference on File and Storage Technologies*. 315–322.
[49] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. 2011. Differentiated storage services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, 57–70.
[50] Microsoft. 2018. Microsoft Azure Storage. Retrieved from https://azure.microsoft.com/en-us/services/storage/.
[51] MPI-forum. 2021. MPI I/O File Info. Retrieved from https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node272.htm.
[52] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2015. Non-volatile storage. *Commun. ACM* 59, 1 (2015), 56–63.
[53] NERSC. 2018. Optimizing I/O performance on the Lustre file system. Retrieved from http://www.nersc.gov/users/storage-and-file-systems/i-o-resources-for-scientific-applications/optimizing-io-performance-for-lustre.
[54] Redhat. 2017. Device-mapper Resource Page. Retrieved from https://www.sourceware.org/dm/.

[55] Redhat. 2017. LVM2 Resource Page. Retrieved from https://sourceware.org/lvm2/.
[56] Redhat. 2018. I/O Limits: block sizes, alignment and I/O hints. Retrieved from https://people.redhat.com/msnitzer/docs/io-limits.txt.
[57] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2018. FStream: Managing flash streams in the file system. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. 257–264.
[58] Seagate. 2017. Enterprise Capacity 3.5 HDD (Helium). Retrieved from http://www.seagate.com/enterprise-storage/hard-disk-drives/enterprise-capacity-3-5-hdd-10tb/.
[59] Milan Shetti, Bingzhe Li, and David Du. 2021. Machine learning-based adaptive migration algorithm for hybrid storage systems. In *IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 202–211.
[60] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2004. Life or death at block-level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Vol. 4. 26–26.
[61] SNIA.org. 2013. SCSI and FC standards update. Retrieved from https://www.snia.org/sites/default/files/files2/files2/SDC2013/presentations/BlockStorage/FredKnight_SCSI_FC_Storage_Standards_Update.pdf.
[62] Technical Committee T10. 2017. SCSI Storage Interfaces. Retrieved from http://www.t10.org/.
[63] T10.org. 2018. SCSI Standards Architecture. Retrieved from https://www.t10.org/scsi-3.htm.
[64] Thomas-Krenn. 2017. Linux Storage Stack Diagram. Retrieved from https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.
[65] Hui Wang and Peter J. Varman. 2014. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the Conference on File and Storage Technologies*. 229–242.
[66] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the Conference on File and Storage Technologies*. 111–118.
[67] Hao Wen, Zhichao Cao, Yang Zhang, Xiang Cao, Ziqi Fan, Doug Voigt, and David Du. 2018. JoiNS: Meeting latency SLO with integrated control for networked storage. In *Proceedings of the IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 194–200.
[68] Hao Wen, David H. C. Du, Milan Shetti, Doug Voigt, and Shanshan Li. 2018. Guaranteed bang for the buck: Modeling VDI applications to identify storage requirements. *IEEE Trans. Cloud Comput.* 9, 2 (2018), 627–640.
[69] Wikipedia. 2021. NVM Express. Retrieved from https://en.wikipedia.org/wiki/NVM_Express.
[70] Fenggang Wu, Ziqi Fan, Ming-Chang Yang, Baoquan Zhang, Xiongzi Ge, and David H. C. Du. 2017. Performance evaluation of host aware shingled magnetic recording (HA-SMR) drives. *IEEE Trans. Comput.* 66, 11 (2017), 1932–1945.
[71] Fenggang Wu, Bingzhe Li, Zhichao Cao, Baoquan Zhang, Ming-Hong Yang, Hao Wen, and David H. C. Du. 2019. ZoneAlloy: Elastic data and space management for hybrid SMR drives. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19)*.
[72] Fenggang Wu, Bingzhe Li, Baoquan Zhang, Zhichao Cao, Jim Diehl, Hao Wen, and David H. C. Du. 2020. TrackLace: Data management for interlaced magnetic recording. *IEEE Trans. Comput.* 70, 3 (2020), 347–358.
[73] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David H. C. Du. 2016. Evaluating host aware SMR drives. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*. USENIX Association.
[74] Fenggang Wu, Baoquan Zhang, Zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and David H. C. Du. 2018. Data management design for interlaced magnetic recording. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*.
[75] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. 2018. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 477–492.

Received January 2021; revised July 2021; accepted September 2021