2018

Host managed storage solutions for Big Data

Pratik Mishra
Iowa State University

Host managed storage solutions for Big Data

by

Pratik Mishra

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Arun K. Somani, Major Professor
Akhilesh Tyagi
Olafsson Sigurdur
Chinmay Hegde
Anuj Sharma

The student author and the program of study committee are solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this dissertation to Late Mr. Vidya Sagar Sharma, my first formal education teacher, who has instilled in me the urge to reason and the faith to believe in hard work right from a very young age. This has always kept me motivated and inspired me to pursue research. My parents, sister, wife, and *Kudzie* have constantly supported, encouraged and provided me strength along my doctoral odyssey. Their sacrifices cannot be measured. I would also like to thank my friends and extended family for their suggestions during the writing of this work. All the teachers, staffs, and people who have taught me lessons in and about life have a vital contribution directly or indirectly. I cannot thank any of you enough in words or action.

"There is no power on earth which can stop an idea whose time has come."— ***Victor Hugo.***

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude towards my doctoral advisor, Dr. Arun K. Somani, for his guidance, patience, and support throughout this research and the writing of this dissertation. Right from my undergraduate junior year in India I have been associated with Dr. Somani. He provided me the opportunity to visit ISU and explore the possibilities to conduct research. His insights and words of encouragement have often inspired me and revived my hopes for completing my doctoral dissertation. He has constantly provided me with personal and professional support during the ups and downs of life. I always had the confidence that I can rely on him for anything and he was always there. The most important lesson from Dr. Somani which I would be carrying throughout life are the values to with-hold as a researcher. These have also made me a better human being. The amount of hard work you have put in is not tangible and words cannot express my indebtedness.

I would like to thank my committee members for their efforts and valuable contributions to this dissertation: Dr. Akhilesh Tyagi, Olafsson Sigurdur, Anuj Sharma, and Chinmoy Hegde. I have had endless hours of discussions and your suggestions which have channelized my thoughts. The guidance you have provided have made my ideas turn into reality. There is no doubt that without your constant support and selfless will, this doctoral research would not have been in the form in which it is today. I take extreme pride that I have been associated to all of you.

I would like to express my special gratitude towards Dr. Mayank Mishra for his valuable suggestions and discussions. In the end, I would also like to thank my lab-mates, the interactions with them, both intellectual and personal were a constant source of encouragement.

Thank-you, Thank-you, and Thank-you. $-$***Fault Tolerance.***

ABSTRACT

The performance gap between Compute and Storage is fairly considerable. With multi-core computing capabilities, CPUs have scaled with the proliferation of Big Data but storage still remains the bottleneck. The physical media characteristics are mostly blamed for storage being slow, but this is partially *true*. The full potential of storage device cannot be harnessed till all layers of I/O hierarchy function efficiently. Despite advanced optimizations applied across various layers along the odyssey of data access, the I/O stack still remains volatile. The problems associated due to the inefficiencies in data management get amplified in multi-tasking Big Data shared resource environments. Its clearly evident that, there is an urgent need to re-think and re-design the system software to address the needs of Big Data.

Software defined storage (SDS) is the means of delivering storage services for a plethora of data center applications and environments. **Our effort is to deliver *near*-ideal performance of storage systems, by identifying issues, designing, and, developing software defined storage capabilities with minimal or no infrastructural change for Data Centers processing Big Data. Thereby, making changes feasible.** We do not intend to change application characteristics or improve storage devices or network infrastructures, but only the way data is managed. Therefore, this research aims to improve the layers along the odyssey of data access environment by understanding the I/O hierarchy and the application needs from storage.

Our contributions have been in three major fields, discussed as follows which are designed and developed specifically to suit multi-tenant, multi-tasking shared Big Data environments.

1) **Operating System optimizations**, deals with optimizing the OS and extending its competency; 2) **Multi-tier solutions** focuses on systems design to incorporate heterogeneous tiers of storage together coupled with value propositions of data being scattered over multiple devices;

3) **Workload specific optimizations** are full-stack data center storage solutions designed and developed to suit workload characteristics.

The Linux OS (host) block layer is the most critical part of the I/O hierarchy as it orchestrates the I/O requests from different applications to the underlying storage. The key to the performance of the block layer is the Block I/O scheduler, which is responsible for dividing the I/O bandwidth amongst the contending processes as well as determines the order and size of requests sent to storage device driver. Irrespective of the data center storage architecture (SAN, NAS, DAS), the final interaction with the physical media is in blocks (sectors in HDD, page in SSD) and the functioning of the block I/O scheduler is highly critical for system performance. Unfortunately, despite its significance, the block layer, essentially the block I/O scheduler hasnt evolved much to meet the needs of Big Data. Due to contention amongst different processes submitting I/O to a storage device and the working of the current I/O schedulers, the inherent sequentiality of MapReduce tasks is lost. This contention causes unwanted phenomenon such as interleaving and/or multiplexing, thereby adversely affecting system performance (CPU wait times, etc.) and increasing latency in disk based (Hard Disk Drive HDDs) storage devices.

First, we develop solutions, BID-HDD, from the core of the operating system, i.e. block I/O scheduling scheme to avoid contentions which tries to maintain the sequentiality in I/O access in order to provide performance isolation to each I/O submitting process and improve individual hard disk drives (HDDs), the details are discussed in Chapter 3. Through trace driven simulation based experiments with cloud emulating MapReduce benchmarks, we show the effectiveness of BID-HDD which results in 28 to 52% lesser time for all I/O requests than the best performing Linux disk schedulers. BID-HDD is essentially a contention avoidance technique which can be modeled to cater different objective functions (storage media type, performance characteristics, etc.). The algorithms developed can be applied to other fields of engineering and science which have time-varying nature of incoming requests and scheduling of events is a challenge (which is the case most often).

HDDs form the back-bone of storage. The physical limitations of HDDs have led to Data Centers organize data in multiple heterogeneous tiers[1] of storage such as those having HDDs and SSDs (Solid State Drives) coupled with workload-aware tiering[2] to achieve cost, performance and capacity trade-offs have become extremely popular. **Second**, we manage multiple devices and develop methodologies, BID-Hybrid, to automated tiering using the information obtained at the block interface using SSDs for improving disk performance (discussed in Chapter 4). BID-Hybrid exploits SSDs random performance to further avoiding contention at disk based storage (using BID). The existing literature tiers based on heuristics or predictions (popularity, frequency, and deviation of logical locations). This may or may not be beneficial and can causes unnecessary deportations to SSD in skewed workload characteristics. BID-Hybrid is a deterministic approach. This enables BID-Hybrid to make judicious decisions and provide a holistic approach to tiering using I/O data-structure information (development of the concepts of *packing fraction*). In our work, we define randomness of blocks usage based on profiling the processes and provide decision metrics based on anticipation and I/O size, in-order to define the correct candidates for tiering. Those blocks are offloaded to SSD which belong to a process causing interruptions (which create *non-bulky* I/Os) and, therefore giving BID-Hybrid the dynamic adaptability based on changing I/O patterns. We demonstrate the effectiveness of BID-Hybrid by using our in-house developed system simulator, with enhanced OS features (VFS and Hybrid-block layer) to realize multiple tiers of storage used together. BID-Hybrid results in performance gain of 6 to 23% for MapReduce workloads when compared to BID-HDD and 33 to 54% over best performing Linux scheduling scheme.

We believe that in a large scale shared production cluster, the issues associated due to data management can be mitigated way higher in the hierarchy of the I/O path, even before requests to data access are made. The current data management techniques fail to capture the syntax and semantics of jobs and the associations of data in various stages of jobs. Moreover, they are mostly reactive and/or based on heuristics or prediction, thereby adding uncertainty. The goals of current

[1]Storage media across all nodes with similar physical and I/O characteristics.

[2]Tiering refers to orchestrating data between heterogeneous tiers of storage by leveraging individual strengths of each to maintain balance between Cost, Performance and Capacity.

efforts have been to make read operations faster as they are believed to be the biggest bottleneck. This problem gets amplified for chained applications which exhibit lineage, where intermediate data during computation must be written and read back later on. Chained Jobs are a popular class of applications that are executed on clusters. Essentially, the jobs are pipelined and the output of a job forms the input (or a part of the input) of the next job. Such jobs are common in several business and scientific applications. The inconsiderate placement of intermediate results (writes) for reuse may affect the read performance adversely. Under this scenario, the gains derived by deploying multiple tiers in storage can be nullified easily by improper replica allocations to tiers, handling of memory resources, and avoidable data movement [Iliadis et al. (2015); Zaharia et al. (2012); Li et al. (2014)]. Therefore, in such data processing pipelines, its imperative to capture lineage or relationships across tasks and their dependency with data, i.e. data-task associations. **Lastly**, we design and develop data management solutions for the complete data center ecosystem using multiple tiers of storage for mitigating the impact of data-dependency in lineage class of applications. LDM, our data management solution, is designed to cater to a class of applications which exhibit **lineage**, i.e. *the current writes are future reads.* In such class of applications, slow writes significantly hurt the over-all performance of jobs, i.e. current writes determine the fate of next reads. The concepts developed can be extended to a wide variety of applications. LDM amalgamates the information from the entire data center ecosystem, right from the application code, to file system mappings, the compute and storage devices topology, etc. to make oracle-like deterministic data management decisions. These policies include, *Data Placement, Replica Management,* and *Data Migration.* With trace-driven experiments, LDM (Algorithms 4 and 5) is able to achieve 29% to 52% reduction in over-all data center workload (lineage as well as other concurrent non-lineage applications) execution time. We believe that LDM will have a huge impact on the performance and resource management of data processing platforms. We discuss briefly the contributions of LDM in workload specific optimization (refer Chapter 5).

With theoretical and experimental evaluations, our host managed storage solutions, namely, BID-HDD, BID-Hybrid, and LDM, fulfils our objective of narrowing the gap between what storage is capable of delivering and what it actually delivers in a Big Data environment.

We conclude our contributions in Chapter 6 followed by brief discussions on the future directions of work in the field of **Host Managed Storage Solutions for Big Data**. In future, we would like to further investigate and develop the field of **"Data assisted systems engineering"**. Throughout this research, we have utilized the information gathered by various layers of the I/O hierarchy to develop storage solutions. Therefore, the data generated from the various components of the system assist in making the performance of Big Data storage systems faster.

Our research would aid Data Centers to achieve their Service Level Agreements (SLAs) as well as to keep the Total-Cost of Ownership (TCO) low. From the Green Computing perspective, our solutions will decrease energy footprint, due to much reduced work to process data across all tiers of computing, i.e. storage, compute (required on storage servers), and network.

CHAPTER 1. THE PROBLEM

1.1 Introduction: Broader Issue

Figure 1.1 represents the progression of computation and storage devices (in terms of speed) over time. Processors have consistently improved and scaled at a steady pace with CPUs processing data at extremely high rates [Nanavati et al. (2015)]. While performance of storage devices remained roughly unchanged for a long time due to physical limitations of mechanical drives. Recently, there have been efforts to reduce the performance gap, right from innovations in the semiconductor industry such as the development of solid state drives to re-inventing faster interconnects such as PCIe to replace the legacy SATA/SAS bus. Despite the best efforts, storage *was, is* and *will* (in the foreseeable future) remain a bottleneck in the system.

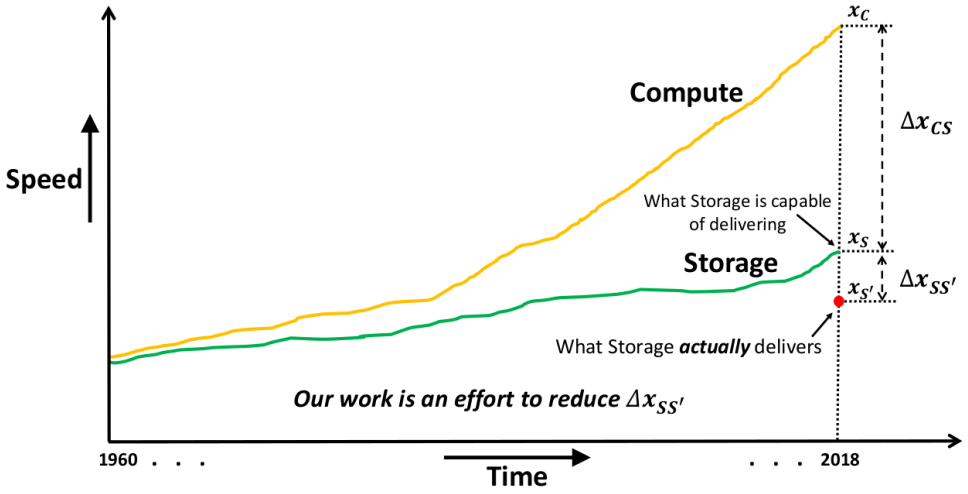


Figure 1.1: Compute and Storage speed progression with time (not to scale).

It is clearly evident that the performance gap (Δx_{CS}) between compute and storage is still fairly considerable. This results in a mismatch between the application needs from storage and what storage can deliver. These two curves depict the *ideal* scenario, i.e. the storage devices deliver

100% of what they are capable of, but practically it never happens. Therefore, *actual* performance difference is much more, i.e. $\Delta x_{CS} + \Delta x_{SS'}$. The additional delay ($\Delta x_{SS'}$) is attributed to what the storage is capable of delivering and what storage *actually* delivers (represented by the red dot).

The physical media characteristics and interface technology are mostly blamed for storage being slow, but this is only partially *true*. The full potential of storage devices (or system) cannot be harnessed till all layers of I/O hierarchy function efficiently. Despite advanced optimizations applied across various layers along the odyssey of data access, the I/O stack still remains volatile. There have been a plethora of solutions to reduce the performance difference, right from OS optimizations like caching, virtualization, pre-fetching, to partitioning of databases, etc., developed to manage data. These solutions have proven to be beneficial for legacy applications with low resource footprint. All these assumptions appear to collapse in Data Centers experiencing Big Data workloads. The problems associated due to the inefficiencies in data management get amplified in multi-tasking, and shared Big Data environments. There is an urgent need to re-think and re-design the system software to address the needs of Big Data.

Our effort is to deliver *near*-ideal performance of storage systems, i.e. reduction of $\Delta x_{SS'}$, by identifying issues and designing storage solutions with minimal or no infrastructural change for Data Centers experiencing Big Data.

In the next section, we briefly discuss the problems associated to data management for Big Data environments, followed by our contributions in Section 1.3.

1.2 Big Data: Associated Issues

Data is growing in an unprecedented rate along all dimensions. Three of the most important V's which create data-intensive workloads is shown in Figure 1.2 [Mishra et al. (2017)]. The sudden spurt in data-driven sciences has put tremendous pressure on the system architecture which was designed for legacy applications with low resource (namely, storage, network and compute)

footprint. For example, to reduce additional seeks to storage, keeping the working set sizes of application small was the key to caching and pre-fetching mechanisms, such that the data could easily fit in RAM. Such techniques and assumptions are now being invalidated due to the current working set sizes and the difficulty in profiling workloads [Harter et al. (2014)].

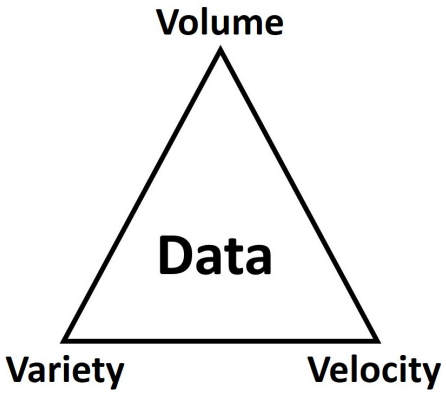


Figure 1.2: Multi dimensional forms of data.

Data centers today cater to a wide diaspora of applications which process multiple data sets for multiple jobs in a multi-user environment concurrently. They also deploy storage systems organized in multiple heterogeneous tiers[1], which is necessary to achieve cost-performance-capacity trade-off [Zhou et al. (2016); Kakoulli and Herodotou (2017)]. Each application can have different syntax and semantics, with varying I/O needs from storage. With highly sophisticated and optimized data processing frameworks, such as Hadoop and Spark, applications are capable of processing large amounts of data at the same time. Dedicating physical resources for every application is not economically feasible [Krish et al. (2014a)]. In cloud environments, with the aid of server and storage virtualization, multiple processes contend for the same physical resource (namely, compute, network, and storage). This causes contentions. In-order to meet their service level agreements (SLAs), cloud providers need to ensure performance isolation guarantees for every application. With multi-core computing capabilities, though CPUs have scaled to accommodate the needs of "Big Data", but storage still remains a bottleneck.

[1]Storage media across all nodes with similar I/O characteristics form a tier.

Given the operating ecosystem, the physical limitations of storage devices and the insatiable needs of applications to process data, the storage layer and the I/O path need to be extremely efficient in-order to minimize delay. The performance of storage devices depend on the order in which the data is stored and accessed. This order is multiplexed due to interferences from other contending applications. Therefore, in large scale distributed systems ("cloud"), data management plays a vital role in processing and storing petabytes of data among hundreds of thousands of storage devices [Zhou et al. (2016)]. Few changes in data-management with proliferation of Big Data is inevitable:

1. The transition to Big Data was sudden and the system software stack was and is not prepared to cope up with the needs of applications from storage. The techniques and schemes used by traditional enterprise infrastructure are being invalidated in highly multiplexing environments such as data centers experiencing Big Data.

2. The current data management techniques fail to capture the syntax and semantics of jobs and the associations of data in various stages of jobs. Under this scenario, the gains derived by deploying multiple tiers in storage can be nullified easily by improper replica allocations to tiers, handling of memory resources, and avoidable data movement. The storage layers needs to be dynamically adaptable to changing time-varying application I/O characteristics.

3. The focus needs to be on Workload-aware tiering[2] coupled with fault-tolerance and data-center topology-awareness to reduce the over-all resource and energy footprints.

4. With so many software and hardware designs, and devices being employed in the data processing infrastructure, a lot of information is generated such as system utilization, device characterization, etc. All such informations should be harnessed to understand the way applications will access data. This could aid data management to dictate deterministic policies and pave the path towards development of **"Data assisted systems engineering"**.

5. Therefore, the entire storage software stack needs to be re-designed with striping up of inefficient layers along the odyssey of data access and adding of new features right from the data center design

[2]Tiering refers to orchestrating data between heterogeneous tiers of storage by leveraging individual strengths of each to maintain balance between Cost, Performance and Capacity.

to operating system kernel I/O sub-structures and network interfaces for matching the application needs from storage with storage capabilities.

Our motivation is based on these assumptions and observations, i.e. the deficiencies of the current storage systems to mitigate the impact of Big Data workloads. In the next section, we briefly outline our contributions in the field.

1.3 Host Managed Storage Solutions: Our Contributions

Software defined storage (SDS) is the means of delivering storage services for a plethora of data center applications and environments. Our major effort has been in developing software defined storage capabilities to manage data with minimal costs and infrastructural changes, thereby making any improvements feasible.

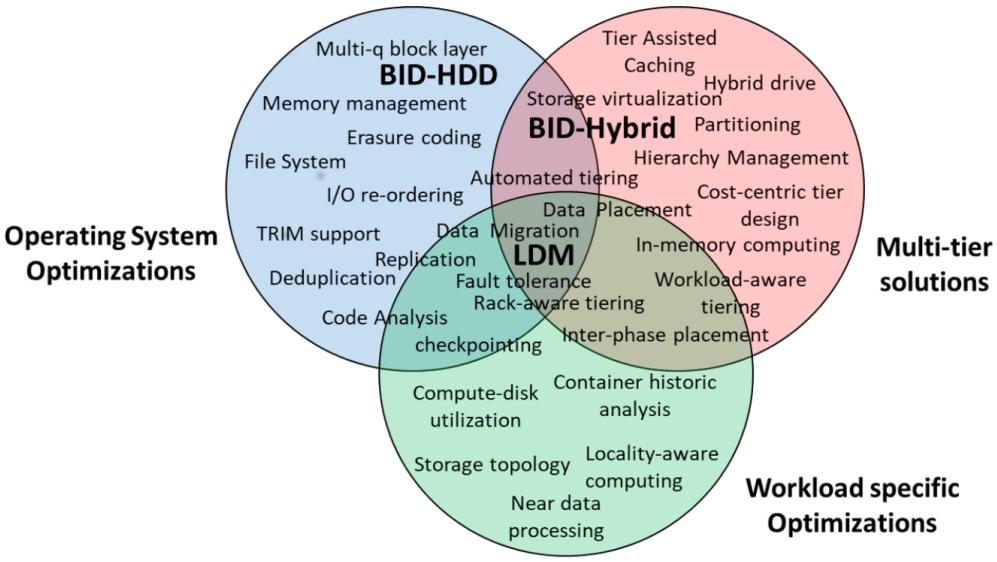


Figure 1.3: Our Contributions in Host Managed Storage.

Our focus is to develop host managed storage solutions by understanding the I/O hierarchy and the application needs from storage for narrowing the gap between what storage is capable of delivering and what it actually delivers in a Big Data environment (refer to Figure 1.1). We do not intend to change application characteristics or improve storage devices or network infrastructures,

but only the way data is managed. Therefore, this research aims to improve the layers along the odyssey of data access environment. Host Managed Storage solutions is a sub-set of software defined solutions which deals with identifying issues, developing and designing software-based data-management storage solutions in the host (operating) side with minimal or no changes to the hardware infrastructure.

Figure 1.3 represents the different areas of research in Host Managed Storage and outlines our contributions in the field. We have three major contributions as outlined below and described in three chapters in this dissertation.

1) **Operating System optimizations** deal with optimizing the OS and extend its competency. We develop solutions, BID-HDD, from the core of the operating system, i.e. a block I/O scheduling scheme to avoid contentions and improve individual storage device (Hard Disk Drives HDDs) capabilities. The details are discussed in Chapter 3.

2) **Multi-tier solutions** focuses on systems design to incorporate heterogeneous tiers of storage together coupled with value propositions of data being scattered over multiple devices.. We manage multiple devices and develop methodologies, BID-Hybrid, to automate tiering utilizing the information obtained at the block interface to use Solid State Drives SSDs for improving disk performance (discussed in Chapter 4).

3) **Workload specific optimizations** are full-stack data center storage solutions designed and developed to suit workload characteristics.We design and develop data management solutions, LDM, for the complete data center ecosystem using multiple tiers of storage to mitigate the impact of data-dependency in lineage class of applications. LDM amalgamates the information from all the strata, devices, and layers in the I/O path (refer to Chapter 5).

All these three categories impact and/or are dependent on each other and many solutions lie in the common areas. Our contributions in each of these chapters are enumerated in the following sections.

1.3.1 Contention Avoidance: OS optimizations

The prime objective of the operating system is to manage data across the I/O hierarchy and relegate the requests from the user-space (or applications) to the storage device. In cloud environments, with the aid of server and storage virtualization, multiple processes contend for the same physical resource (namely, compute, network, and storage). This causes *contentions*. In-order to meet their service level agreements (SLAs), cloud providers need to ensure performance isolation gaurantees for every application. The performance of storage devices depend on the order in which the data is stored and accessed. This order is multiplexed due to interferences from other contending applications. Therefore, in such systems, data management plays a vital role in processing and storing petabytes of data among hundreds of thousands of storage devices. Therefore, optimizing the operating system is extremely critical for the over-all system performance.

We enumerate our contributions in optimizing the operating system as follows.

- Identifying the major source of contention in the I/O subsystem, i.e. "request queue processing" (refer to Appendix A).

  The Linux OS (host) block layer is the most critical part of the I/O hierarchy, as it orchestrates the I/O requests from different applications to the underlying storage. Unfortunately, despite its significance, the block layer, essentially the block I/O scheduler has not evolved to meet the needs of Big Data.

- Identification of the requirements of a block I/O scheduler suited for Big Data environments.

  The data access time in HDDs is majorly governed by disk arm movements, which usually occurs when data is not accessed sequentially. Big Data applications exhibit evident sequentiality but due to the contentions amongst other I/O submitting applications, the I/O accesses get multiplexed which leads to higher disk arm movements. The requirements are laid down in Section 3.3.

- We have developed and designed a contention avoidance scheme for disk based storage devices known as "BID-HDD: Bulk I/O Dispatch" in the Linux block layer, specifically to suit multi-tenant, multi-tasking and skewed shared Big Data deployments.

- BID-HDD extends the capabilities of the current block layer to adapt with changing Big Data workloads.

  The efficient pipelining of large data blocks groups from adjoining locations in the disk leads to reduction in disk arm movements (leveraging sequentiality performance). The development of staging queues as well as the scheduling algorithms (Algorithms 1 and 2) has enabled the block layer to make judicious decisions of dispatching requests to the device driver. The dynamic need-based anticipation time ensures performance isolation to each I/O contending processing following system constraints without compromising the SLAs.

- We have designed and developed a System Simulator using Python v2.7.3 to replicate the working of the System level components (Host OS, Storage devices, etc.).

  We use the trace file (as discussed in Section 3.6.1) for application I/O submission order for evaluating our system design. The OS simulator is designed to work right from the kernel I/O data-structures and development of these units. While the storage simulator is accurately developed to emulate the working of the storage devices as per device characteristics and specifications.

- Through trace driven simulation based experiments with cloud emulating MapReduce benchmarks, we show the effectiveness of BID-HDD which results in 28 to 52% lesser time for all I/O requests than the best performing Linux disk schedulers.

BID-HDD is essentially a contention avoidance technique which can be modeled to cater different objective functions (storage media type, performance characteristics, etc.). The algorithms developed can be applied to other fields of engineering and science which have time-varying nature of incoming requests and scheduling of events is a challenge (which is the case most often). BID-HDD and the associated details are discussed in Chapter 3.

1.3.2 Contention Avoidance: Multi-tier solutions

Due to physical limitation of HDDs, there have been recent efforts to incorporate flash based high-speed, non-volatile secondary memory devices, known as Storage Class Memories (SCMs) in data centers. Despite superior random performance of SCMs (or SSDs) over HDDs, replacing disks with SCMs completely for data center deployments does not seem to be economically feasible. With recent developments in NVMe devices, with supporting infrastructure and virtualization techniques, a hybrid approach of using heterogeneous tiers of storage together such as those having HDDs and SSDs coupled with workload-aware tiering to balance cost, performance and capacity have become increasingly popular.

Data centers consists of many tiers of storage devices. All storage devices of the same type form a tier. For example, all HDDs across the data-center form the HDD tier and all SSD form SSD tier, and similarly for other SCMs. Based on profiling of workloads, balanced utility value of data usage, the data is managed between the tiers of storage for improved performance. Workload-aware storage tiering, or simply *tiering* is the automatic classification of how data is managed between heterogeneous tiers of storage in an enterprise data-center environment [Mishra and Somani (2017)]. It is vital to develop automated and dynamic tiering solutions to utilize all the tiers of storage. Our contributions in multi-tier OS contention avoidance storage solutions are described below.

- We develop and design a hybrid scheme, BID-Hybrid, to exploit SCMs (SSDs) superior random performance to further avoid contentions at disk based storage to suit such multi-tasking, multi-user shared Big Data environments. BID-Hybrid aims to deliver the capability of dynamic and judicious automated tiering in the block layer as a SDS solution.

- BID-Hybrid lies in the "initial tier placement" class of problem in tiering. The main objective function of "initial tier placement" problem is the balanced decision of which tier the data is to be initially written in-order to reap the maximum performance benefits.

- Contrary to the tiering approach of defining SSD candidates based on deviation of LBAs, BID-Hybrid profiles process I/O characteristics by utilizing dynamic anticipation and I/O packing of kernel data-structures.

- BID-Hybrid is able to efficiently offload non-bulky interruptions from HDD request queue to SSD queue using BID-HDD for disk request processing and multi-q FIFO architecture for SSD.

- We design the system architecture to support a "hybrid OS block layer" (see Section 4.4) and develop system simulators to evaluate BID-Hybrid.

- BID-Hybrid results in performance gain of 6 to 23% for MapReduce workloads when compared to BID-HDD and 33 to 54% over best performing Linux scheduling scheme. BID schemes as a whole is aimed to avoid contentions for disk based storage I/Os following system constraints without compromising SLAs.

BID-Hybrid (refer to Chapter 4) uses similar concepts of staging as BID-HDD. Due to the staging capabilities in the Host (OS) block layer, bulkiness of processes can be calculated and verified on-the fly in-order to avoid unnecessary deportations to SSD. The key idea is to offload I/O blocks belonging to non-bulky processes to SSD (managed by multi-q block layer architecture [Bjørling et al. (2013)]) and the bulky I/Os to HDD (handled by BID-HDD). This serves multi-fold: (1) maximal sequentiality in HDD is ensured, i.e "HDD request queue" is made free from unnecessary contention and interruption causing blocks; (2) the future references to the non-bulky blocks are prevented from causing contentions for HDD disk I/O, as the semantic blocks have a high probability to appear in the same pattern. Therefore, BID-Hybrid aims to further reduce contention (more than BID-HDD) at disk based storage by offloading interruption causing blocks to SSD, while ensuring uninterrupted sequential access to HDDs.

1.3.3 Workload Specific Optimizations: Exploring Lineage

Data centers today cater to a wide diaspora of applications which process multiple data sets for multiple jobs in a multi-user environment concurrently. They also deploy storage systems organized in multiple heterogeneous tiers, which is necessary to achieve cost-performance-capacity trade-off [Mishra and Somani (2017); Iliadis et al. (2015); Kim et al. (2011)]. Dedicating physical resources for every application is not economically feasible. Resource sharing causes contention affecting the efficiency and performance [Mishra and Somani (2017); Mishra et al. (2016); Hindman et al. (2011)]. Data are scattered over multiple files located at multiple storage nodes[3] and replicated for performance, availability and reliability reasons.

We believe that in a large scale shared production cluster, the issues associated due to data management can be mitigated at a much higher level in the hierarchy of the I/O path, even before requests to data access are made. The current data management techniques fail to capture the syntax and semantics of jobs and the associations of data in various stages of jobs. Moreover, they are mostly reactive and/or based on heuristics or prediction, thereby adding uncertainty. Moreover, the goals of current efforts have been to make read operations faster as they are believed to be the biggest bottleneck. This problem gets amplified for chained applications which exhibit lineage, where intermediate data during computation must be written and read back later on. For example, Chained Jobs are a popular class of applications that are executed on clusters. Essentially, the jobs are pipelined and the output of a job forms the input (or a part of the input) of the next job. Such jobs are common in several business and scientific applications.

The inconsiderate placement of intermediate results (writes) for reuse may affect the read performance adversely. It is now becoming clearer that dealing with large amounts of current "writes", which are future "reads" is equally important to achieve good performance. Under this scenario, the gains derived by deploying multiple tiers in storage can be nullified easily by improper replica allocations to tiers, handling of memory resources, and avoidable data movement [Iliadis et al. (2015); Zaharia et al. (2012); Li et al. (2014)]. Therefore, in such data processing pipelines, it

[3]Storage refers to the overall data plane, whereas a storage node refers to a single physical device.

is imperative to capture lineage or relationship across tasks and their dependency with data, i.e. data-task associations.

LDM is designed to cater to a class of applications which exhibit **lineage**, i.e. *the current writes are future reads*. In such class of applications, slow writes significantly hurt the over-all performance of jobs, i.e. current writes determine the fate of next reads. The concepts developed can be extended to a wide variety of applications. We discuss briefly the contributions of LDM in workload specific optimization (refer to Chapter 5).

- We develop and design a novel framework, called **LDM**, to address the challenges in lineage-aware data management to effectively utilize multi-tier storage hierarchy. LDM captures the inherent lineage information and reduce the data movement via network by placing them appropriately to enable maximal processing nearer to the storage locations as well as in appropriate storage tiers.

- LDM amalgamates the information from the entire data center ecosystem, right from the application code, to file system mappings, the compute and storage devices topology, etc. to take oracle-like deterministic data management decisions.

- LDM captures the inherent data dependency by analyzing the metadata associated with application code and extract semantic knowledge of the computational workflow logic coupled with the file system information to build task and block graphs.

- We develop *block-graphs*, which uses file-system information about the block to device mappings to associate blocks of data with tasks (using task blocks). Block graphs are designed to deterministically capture all the data block-task associations and data lineage across tasks. LDM uses this knowledge to mitigate the impact of delays associated to writing and then subsequently reading intermediate results.

- LDM utilizes all tiers of storage to reduce data access delays in conjunction with workload aware tiering by orchestrating multiple data management features. LDM takes into account

the storage device current and future utilization along with its characteristics and match them to "lineage-quotient" of blocks to dictate policies.

- LDM performs *Initial Data Placement, Replication Placement,* and *Data Migration* tasks for dependency mitigation which are described using Algorithm 4, 5 and 6, respectively. They determine the storage device(s) to place the data and if, when, and where to move data blocks dynamically.

- With trace-driven experiments, we show LDM (Algorithms 4, and 5) is able to achieve 29% to 52% reduction in over-all data center workload execution time.

We believe that LDM will have a huge impact on the performance and resource management of data processing platforms.

From the Green Computing perspective, our solution will decrease energy footprint, due to much reduced work to process data across all tiers of computing, i.e. storage, compute (required on storage servers), and network.

1.4 Dissertation Organization

In Chapter 1, we have briefly described the problems associated to storage and deficiency of the traditional (current) system architecture to brave with the requirements of Big Data. This is followed by the description of the problem and our major contributions in the field. Chapter 2 categorizes the relevant literature in our problem scope, and provides an overview of the work done in developing these areas, namely, OS optimizations, Multi-tier OS solutions, and Workload specific optimizations, respectively.

The next three chapters are organized based on our specific contributions. Chapter 3 describes our novel contention avoidance technique, BID-HDD, which essentially is a block I/O scheduler for disk based storage suited for multi-tenant, multi-user, and multi-tasking Big Data shared resource environments. We discuss right from the details of the operating system basics and develop the solution from the core to match the current application needs from storage. In Chapter 4, we

discuss the working architecture of our novel multi-tier (HDDs and SCMs) OS contention avoidance scheme, BID-Hybrid, which further reduces contentions for disk based storage devices using solid state devices. Chapter 5 describes the working of our data management solution, LDM, designed to cater a class of applications which exhibit lineage, i.e. *the current writes are future reads*. In such class of applications, we mitigate the impact of slow writes which significantly hurt the over-all performance of jobs, i.e. current writes determine the fate of next reads.

We conclude our research in Chapter 6, followed by brief discussions on the future directions of work in the field of **Host Managed Storage Solutions for Big Data**.

CHAPTER 2. LITERATURE REVIEW

The journey to store and utilize data from cave drawings to developments in exascale storage has fueled various innovations over time. The main research focus for a long time has been in improving physical media characteristics like increasing areal density of hard drives, read/write technology, etc., semiconductor storage devices (SSDs, DVIMMs, RAMDisks, etc.), and their interconnects (SAS, SATA, PCIe, NVMe, FCoE etc.). The developments in the field of storage can be broadly divided into application-space, networking and storage-devices. We limit our discussion to our problem scope, i.e. **Host Managed Storage Solutions for Big Data.**

2.1 Our Problem scope

Our effort is to provide software defined storage capabilities by identifying and eliminating the deficiencies along the software layers of the I/O path for data-centers experiencing Big Data workloads. We have categorized the relevant literature according to the specific areas of our problem scope, as shown in Table 2.1.

2.1.1 Block I/O Optimizations

In this section, we discuss the developments in the block layer, concentrating mostly on I/O Scheduling. I/O Scheduling has been around since the beginning of disk drives [Ruemmler and Wilkes (1994)]. We limit our discussion to those approaches which are relevant to recent developments. Despite advanced optimizations applied across various layers along the odyssey of data access, the Linux I/O stack still remains volatile. The block layer hasn't evolved [Bjørling et al. (2013); Riska et al. (2007)] to cater the requirements of Big Data. Riska et al. (2007) evaluates the effectiveness of block I/O optimization at the application layer by quantifying the effect of request merging and reordering at different I/O layers (File System, Block Layer, Device driver) have on

overall system performance. One of the major findings were in establishing relationships between performance and block I/O scheduler. Our work on BID-HDD is an effort in this domain especially for rotation based recording drives. BID is essentially a contention avoidance technique which can be modeled to cater different objective functions (storage media type, performance characteristics, etc.).

Axboe (2004) provides a brief overview of the Linux block layer, basic I/O units, request queue processing, etc. Ibrahim et al. (2011) proposes a framework which studies the VM interference in Hadoop virtualized environments with the execution of single MapReduce job with several disk pair schedulers. It divides the MapReduce job into phases (i.e. Map, Shuffle, and Reduce) and executes series of experiments using a heuristic to choose a disk pair scheduler for the next phase in a VM Environment. Bhadkamkar et al. (2009) is a self-optimizing HDD based solution which re-organizes blocks in the block layer by forming sequences via calculating correlation amongst LBA (logical block address) ranges with connectivity based on frequency distribution and temporal locality. It makes weighted graphs and relocation of blocks happens to most needed vertex first. The goal is to service most requests from dedicated zones of a HDD.

Bjørling et al. (2013) is an important piece of work which extends the capabilities of the block layer for utilizing internal parallelism of SSDs to enable fast computation for multi-core systems. It proposes changes to the existing OS block layer with support for multiple software and hardware queues for a single storage device. Multi-q involves a software queue per CPU core. Similar lock-contention scheme can be used for BID, as it also involves multiple queues. Malladi et al. (2016) mentions about NVMe I/O scheduling having separate I/O queues for each core, therefore using Multi-q concepts. In BID-Hybrid, we use Multi-q for serving I/Os in SSD as it would ensure performance as well as allow proportional sharing.

Yi et al. (2017) is an SSD extension of CFQ scheduler in which each process has a FIFO request queue and the I/O bandwidth is fairly distributed in round robin fashion. Park et al. (2016) and Kim et al. (2016) propose to ensure diverse SLAs, including reservations, limitations, and proportional sharing by their I/O Scheduling schemes in shared VM environment for SSDs.

While Park et al. (2016) uses an opportunistic goal oriented block I/O scheduling algorithm, Kim et al. (2016) proposes host level SSD I/O schedulers, which are extensions of state-of-the-art I/O scheduling scheme CFQ. Wang et al. (2013) tries to utilize the parallelism in SSDs, by dividing the entire SSD into sub-regions, each having a different queue for dispatching requests. Wang et al. (2013) might be good in applications which have more random I/Os otherwise, leading to increasing wait queues for popular sub-regions & bias in performance.

2.1.2 Multi-tier solutions

There is a huge industrial and academic focus to incorporate NVMe's (SSDs) into data-centers, with developments such as NVMe Express utilizing PCIe bus technology and NVMe over RDMA Fabrics for point-to-point interconnect [Nanavati et al. (2015); Malladi et al. (2016)]. Though hard drives will not be replaced by NVMe devices (SSDs) in the near future, more prominently due to SSD's high TCO (Total Cost of Ownership- cost/GB, write amplification, lifespan) [Yang and Zhu (2016b)], lack of consistent software stack (fabrics, interface and media characteristics) as well as non-uniform workload performance characteristics [Nanavati et al. (2015); Mittal and Vetter (2016); Krish et al. (2016)]. A hybrid approach with heterogeneous tiers of storage such as those having HDDs and SCMs coupled with workload aware tiering to balance cost, performance and capacity have become increasingly popular [Zhou et al. (2016); Harter et al. (2014)]. Multi-tier storage environment deal with how data is managed between heterogeneous tiers of storage in enterprise data-center environment.

The underlying foundation of multi-tier storage has been adopted from the concepts of caching mechanisms such as LRU, LFU, etc., as well as partitioning of databases. Partitioning of databases, more specifically vertical partitioning has been an active field of research since the 70's and 80's [Galaktionov et al. (2016); Li and Patel (2014)]. The key idea is to develop an optimization model to satisfy one or more criteria to improve the I/O performance of databases. Partitioning of databases, similar to physical design problems has been proven to be NP-Hard due to the estimation errors in both system and workload parameters [Galaktionov et al. (2016); Li and Patel (2014); Agrawal

et al. (2004)], therefore extensive work has been done by the database community [Navathe et al. (1984); March and Rho (1995); Chu (1969); Cornell and Yu (1990); Alagiannis et al. (2014); Curino et al. (2010); Jindal and Dittrich (2011); Jindal et al. (2013); LeFevre et al. (2014)].

Navathe et al. (1984), Cornell and Yu (1990), March and Rho (1995) & Chu (1969) have been one the earliest studies in the filed of partitioning of databases. Navathe et al. (1984) proposed algorithms and physical system designs to vertically partition databases to reorganize data in two level memory hierarchy such that highly active data is stored in the fastest memory. This is done to minimize the access to secondary storage, thereby improving performance. Chu (1969) developed an optimization model for minimizing overall costs by constricting response time and capacity with fixed number of copies of each file fragment. Cornell and Yu (1990) proposes a data allocation strategy to optimize performance of distributed databases. Their solution has a major limitation as they assume the network to be fully connected with each link having equal bandwidth. March and Rho (1995) proposed a comprehensive genetic algorithm based model to allocate operations to nodes taking into consideration replication and operation allocation costs.

All these previous studies by the database community were based on static workloads, which restricts their use for constantly changing workloads [Galaktionov et al. (2016); Li and Patel (2014)]. Dynamically adaptive variations of these concepts have been explored thoroughly in the design of modern datastores [Galaktionov et al. (2016); Li and Patel (2014); Curino et al. (2010)]. These methods are used in online data partitioning such as O2P, H20 [Alagiannis et al. (2014)], etc., and disk based analytical databases [Li and Patel (2014); He et al. (2011); Jindal et al. (2013)]. In the Big Data ecosystem, most prominently the concepts of Navathe et al. (1984); Cornell and Yu (1990); Chu (1969); March and Rho (1995) have laid strong footing for the data layout design on HDFS [Li and Patel (2014); He et al. (2011); Jindal et al. (2011)]. Another use case has been in tuning of data stores [Galaktionov et al. (2016); Guo et al. (2012)], such as a multi-store with HDFS and RDMS together, where every parameter of the datastore is not known apriori. Integration of both horizontal and vertical partitioning together [Agrawal et al. (2004)] have led to the design of modern Column stores and NoSQL datastores, most popularly, Hbase [Harter et al. (2014)], WideTable [Li

and Patel (2014)], RCFile [He et al. (2011)], etc. There has been a lot of prior work done on caching and partitioning, which are the predecessors of multi-tier storage. We focus our attention towards recent developments in multi-tier storage solutions which involve data management between storage devices such as HDDs and SSDs.

Most of the literature in multi-tier storage solutions has concentrated on finding the temperature of data, and migrating "hot data" form slower HDD tier to SSD tier and vice versa for "cold" data. The effects of caching in enterprise platforms in negligible due to the data set size and skewed workload characteristics [Harter et al. (2014); Krish et al. (2013)], therefore faster SCMs (SSDs) are used as cache. Zhang et al. (2010) proposes an adaptable data migration model based on the heat of data to determine the next hot data. Lin et al. (2011) migrates or allocates files to SSD based on hotness (access frequency), randomness and profit-value based on read/write-intensiveness and recency of file access. Chang et al. (2015) keeps blocks in SSD with highest hit frequency. Migration is based on utility value associated with every block in SSD in last time slot based on read/write counts, known as profit caching. Hybrid-disk Aware CFQ scheduling proposed is an extension of CFQ in which the I/O's to SSD are serviced immediately. Ye et al. (2015) proposes a time-decay regional popularity replacement algorithm for blocks with high probability of being popular and migrate them from HDD to SSD. Regions are adjacent blocks in HDD. Though multiple efficient techniques have been proposed, in shared Big Data cloud deployments due to the highly skewed, non-uniform and multiplexing workloads [Ibrahim et al. (2011)], prediction of utility value of blocks for tiering based on heat of data might not be a viable option.

Our proposed solution BID-Hybrid, however, lies in the "initial tier placement" problem, in which the goal is to decide which tier the data is to be written in-order get maximum performance benefits. While BID-Hybrid works on the principle of making judicious, anticipated and dynamic tier placement decision based on bulkiness of processes, non-bulky data is offloaded to SSD and bulky in HDD. This serves multi-fold, first ensuring uninterrupted sequential data access on HDDs. Secondly, preventing performance critical future interruptions in HDDs. These semantic blocks

which are non-bulky are offloaded to SSDs have a high probability to appear in the same pattern [Bhadkamkar et al. (2009); Ibrahim et al. (2011); Chen et al. (2011)].

In the existing literature tiering is based on randomness in I/O, and is defined as mere deviation of LBA (logical block addresses). An application could be sequential but due to contention at the request queue to submit requests may appear as random in such a case. This might thereby causes unnecessary deportations to SSD in skewed workload characteristics. In BID-Hybrid, we take care of such cases and define randomness for blocks based on profiling the processes and provide decision metrics based on anticipation and I/O size, in-order to define the correct candidate for tiering. Therefore, BID-Hybrid uses the notion of randomness of process characteristics to make dynamic and judicious tier-placement decisions.

PASS involves high cost due to retiring SSDs (limited write/erase cycles) with lack of workload-aware tiering, i.e. SSD is used as absorption layer, which wont be suitable for skewed workloads like MapReduce. Iliadis et al. (2015) determines data-to-tier assignments for Data-Centers based on cost-function (based on chunk size/request size, rate, volume of storage) to reduce mean response time. It simulates the inter-arrivals as a M/G/1 single server queue and processing is done as per chunk size. Kim et al. (2011) proposes a tool for improving capacity planning within cost-budgets & performance guarantees during deviations from expected workloads. Krish et al. (2016) studies HDFS characteristics to place intermediate data of MapReduce in SSD to improve performance and cost-optimization. Many MapReduce workloads have large and sequential intermediate data sets, SSDs could be a bottleneck.

Shi et al. (2012) computes optimal data file by creating a multi-choice 0/1 Knapsack problem to reduce number of transfers between tiers for data allocation. I/O information from clients are used to distinguish sequential and random. Random and hot objects are allocated to tiers according to the Knapsack problem. In Liu et al. (2010), SSD is split into Read and Write cache. The I/O operations are monitored in the OS Kernel. The Dispatcher module detects sequentiality from the "request queue" by the number of continuous LBAs. The random blocks are recorded in a table and data is cached in read cache of SSD. When a page is evicted from Page Cache, its LBA is

checked in the table and if a hit is found, it caches data in read cache. Migration follows LFU, when the utilization rate of read cache is 90%. Krish et al. (2014b) redesigns HDFS for a multi-tiered hybrid storage based on tier characteristics and capacity. It logically groups all storage devices in a tier across all nodes and manages them individually. It increases utilization of HPC storage by forwarding greater number of I/Os to faster tiers and exploits tier information to decide where to place replicas of a block. Islam et al. (2015) designs a hybrid storage for HPC including RAM disks, SSDs, HDD and utilize Lustre FS and HDFS. It deals with tri-replication of blocks ensuring fault tolerance. The data placement decision is based on storage space available and migration from layer to layer is based on the priority of usage.

2.1.3 Workload Specific Optimizations: Exploring Lineage

Most studies have focused on studying data center operations to consolidate the computing needs and organize and optimize computing for multiple applications. Computing resources are believed to be abundant, but without appropriate attention, they are mostly waiting for data and wasting cycles [Mishra and Somani (2017); Bjørling et al. (2013); Bhadkamkar et al. (2009); Choi et al. (2017); Bu et al. (2010); Afrati and Ullman (2010)]. Moreover, for lineage based applications, the impact is more severe due to data-dependency between tasks. Keeping all the data in memory (as done in Spark) may not be a wise choice either. We believe that the focus needs to shift from computing to data. What makes this shift relevant is the availability of oracle-like deterministic workload and data center storage topology aware data management. Datum access from storage and copying in memory is expensive. Therefore, we believe that studying data utilization patterns and developing strategies to optimize computing paths are the greatest needs at the current time [Zaharia et al. (2010)].

There have been efforts [Li et al. (2014); Zaharia et al. (2012)] to understand lineage for in-memory computation for improving job recovery time in-case of fail-overs and performance in Data Centers with nodes having large memory. Zaharia et al. (2012) forms distributed data-sets for in-memory computations (production and computation in-memory), which inherently improves

performance. Li et al. (2014) proposes an in-memory fault tolerant mechanism which leverages lineage to recover lost outputs by re-executing the steps which formed the data-sets. In-memory computations and storing of results in memory are infeasible for Big Data workloads as the working sets are huge to fit in RAM, along-with the time-varying nature of applications for production and consumption of data blocks [Harter et al. (2014)]. Issues such as ensuring reliability and cost-effectiveness are other major challenges in such frameworks. Therefore, cost simulations in Harter et al. (2014) that adding small SCM tier and efficient orchestrating data between tiers can lead to enhanced performance than equivalent spending on RAM or disks. Multi-tier storage offers multiple dimensions, such as device type, network connectivity, and replication management, which allows to explore to explore the issues associated with data access differently.

Multiple solutions [Kakoulli and Herodotou (2017); Grund et al. (2010); Islam et al. (2016); Krish et al. (2014b); Lee et al. (2016); Gunda et al. (2010); Olson et al. (2017); Zhang et al. (2010); Mihailescu et al. (2012); Ananthanarayanan et al. (2012); Iliadis et al. (2015); Islam et al. (2015); Grund et al. (2010)] have been proposed in literature to exploit multi-tier storage, but none addresses the issues associated with lineage or chained jobs.

Islam et al. (2015) designs a heterogeneous storage engine for HPC including RAMdisks, SSDs and HDDs, and Lustre FS to benefit HDFS. The data placement engine in Islam et al. (2015) deals with tri-replication of blocks to ensure fault tolerance and the decisions of placement of replicas in a tier is based on storage space available with a usage-priority based tier migration model. Krish et al. (2014b) proposes a model with an intent to remove performance bottlenecks by placing every block belonging to file in all tiers of storage.

In the following chapters, we discuss our host managed storage solutions, namely, BID-HDD, BID-Hybrid, and LDM.

Table 2.1: Related Works Categorization.

Block I/O Optimizations	**Multi-tier Solutions**	**Workload Specific**
Bjørling et al. (2013), Yi et al. (2017) , Park et al. (2016), Axboe (2004), Xu et al. (2015), Chen et al. (2011), Chang et al. (2015), Wang et al. (2013), Ji et al. (2017), Mandal et al. (2016), Valente and Andreolini (2012), Malladi et al. (2016), Ibrahim et al. (2011), Bhadkamkar et al. (2009), Galaktionov et al. (2016), Alagiannis et al. (2014), Olson et al. (2017), Iyer and Druschel (2001), Vangoor et al. (2017), Zheng et al. (2013), Bisson and Brandt (2007), Yang and Zhu (2016a), Riska et al. (2007), Mittal and Vetter (2016), Roussos (2007), Pfefferle et al. (2015), Lee et al. (2017) Yang and Zhu (2016b), Koller and Rangaswami (2010), Joo et al. (2017), Kim et al. (2016), Ma and Xu (2016), Aghayev et al. (2017), Park and Shen (2012).	Zhang et al. (2010), Zhou et al. (2016), Krish et al. (2016), Islam et al. (2016), Krish et al. (2014b), Chang et al. (2015), Lin et al. (2011), Ye et al. (2015), Liu et al. (2010), Xiao et al. (2012), Islam et al. (2015), Tiwari et al. (2012), Herodotou (2016), Iliadis et al. (2015), Kim et al. (2011), Krish et al. (2013), Krish et al. (2014a), Chen et al. (2011), Khasnabish et al. (2017), Bisson and Brandt (2007), Moon et al. (2015), Ananthanarayanan et al. (2012), Kakoulli and Herodotou (2017), Xu et al. (2015), Olson et al. (2017), Borba and Tavares (2017), Kang et al. (2013), Shi et al. (2012), Lu et al. (2017), Spivak et al. (2017), Jindal et al. (2013), Ma and Xu (2016), Curino et al. (2010), Krish et al. (2016).	Li et al. (2014), Zaharia et al. (2012), Ananthanarayanan et al. (2012), Bu et al. (2010), Gunda et al. (2010), Lu et al. (2017), Jahani et al. (2011) , Choi et al. (2017), Hindman et al. (2011), Lee et al. (2016), Joo et al. (2017), Gkantsidis et al. (2013), Kakoulli and Herodotou (2017), Curino et al. (2010), Guo et al. (2012), Jindal et al. (2011), Ferdaus et al. (2017), Gao et al. (2017), Krish et al. (2013), Mihailescu et al. (2012), Zaharia et al. (2010), Grund et al. (2010), Xie et al. (2011), Shi et al. (2012), Li and Patel (2014), Tiwari et al. (2012), Islam et al. (2016), Riedel et al. (1998), Lee et al. (2016), Afrati and Ullman (2010), Abad et al. (2012), LeFevre et al. (2014), Moon et al. (2015), Harter et al. (2014)

CHAPTER 3. CONTENTION AVOIDANCE FOR DISK BASED BIG DATA STORAGE

In this chapter, we discuss the details and impact of our work on managing I/O contentions in the operating system for disk based storage devices deployed in data centers experiencing Big Data workloads.

The chapter is organized as follows. First, we discuss *contentions* and the associated issues in Section 3.1. Section 3.2 provides a brief overview of the working of the I/O stack, HDD characteristics and its inefficiencies in shared large data processing infrastructure. Section 3.3 lays down the expectation from a block I/O scheduler in Big Data deployments as well as points out the issues with the current Linux scheduling schemes. In Sections 3.5 and 3.6, we present our Contention Management scheme i.e. block I/O scheduler, **BID-HDD** [Mishra et al. (2016)] followed by our design of experiments and performance evaluation, respectively. Section 2.1.1 discussed in Chapter 2 provides an in-depth survey of related literature. We conclude the chapter in Section 3.7 with a discussion on future work.

3.1 The Problem

Data Centers today cater to a wide diaspora of applications, with workloads varying from data science batch and streaming applications to decoding genome sequences. Each application can have different syntax and semantics, with varying I/O needs from storage. With highly sophisticated and optimized data processing frameworks, such as *Hadoop* and *Spark*, applications are capable of processing large amounts of data at the same time. Dedicating physical resources for every application is not economically feasible [Krish et al. (2016)]. In cloud environments, with the aid of server and storage virtualization, multiple processes contend for the same physical resource (namely, compute, network and storage) [Kim et al. (2016)]. This causes *contentions*. In-order to meet their

Service Level Agreements (SLAs), cloud providers need to ensure performance isolation gaurantees for every application [Nanavati et al. (2015)].

With multi-core computing capabilities, CPUs have scaled to accommodate the needs of *"Big Data"*, but storage still remains a bottleneck. The physical media characteristics and interface technology are mostly blamed for storage being slow, but this is partially *true*. The full potential of storage devices cannot be harnessed till all the layers of the I/O hierarchy function efficiently. The performance of storage devices depend on the order in which the data is stored and accessed. This order is multiplexed due to interferences from other contending applications. Therefore, in large scale distributed systems ("cloud"), data management plays a vital role in processing and storing petabytes of data among hundreds of thousands of storage devices [Zhou et al. (2016)]. The problems associated due to the inefficiencies in data management get amplified in multi-tasking, and shared Big Data environments.

Big Data applications use data processing frameworks such as *Hadoop MapReduce*, which access storage in large data chunks (64/128 MB HDFS blocks), therefore exhibiting evident sequentiality. Due to contentions amongst concurrent I/O submitting processes and the working of the current I/O schedulers, the inherent sequentiality of Big Data processes is lost. These processes may be instances of the same application (Map, shuffle or reduce tasks) or belong to other applications. The contentions result into unwanted phenomenons such as multiplexing and interleavings, thereby breaking of large data accesses [Joo et al. (2017); Yi et al. (2017); Park et al. (2016)]. The increase in latency of storage devices (HDDs) adversely affects overall system performance (CPU wait time increase) [Bhadkamkar et al. (2009)].

Despite advanced optimizations applied across various layers along the odyssey of data access, the I/O stack still remains volatile. The Linux OS (Host) block layer is the most critical part of the I/O hierarchy as it orchestrates the I/O requests from different applications to the underlying storage. The key to the performance of the block layer is the Block I/O scheduler, which is responsible for dividing the I/O bandwidth amongst the contending processes as well as determines the order of requests sent to storage device. Figure 3.1 shows the importance of the block layer.

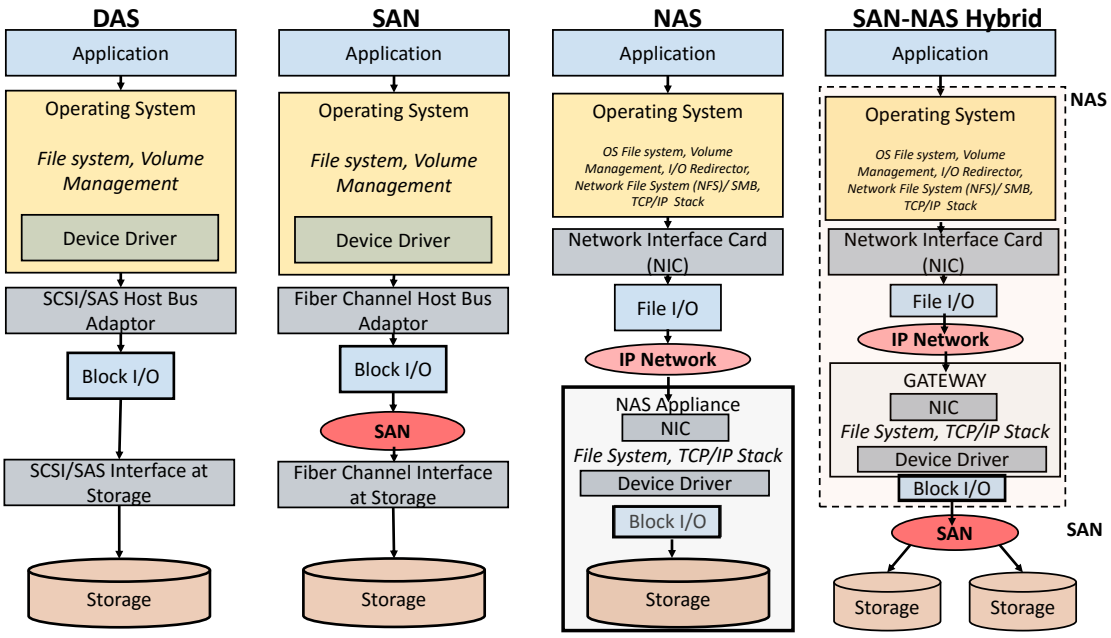


Figure 3.1: Networked Storage Architecture.

We observe that irrespective of the data-center storage architecture, i.e. SAN, NAS or DAS, the final interaction with the physical media is in blocks (sectors in HDD, pages in SSD). The block layer is employed to manage I/Os to the storage device. Hard Disk Drives (HDDs) form the backbone of data center storage. The data access time in HDDs is majorly governed by disk arm movements, which usually occurs when data is not accessed sequentially. Big Data applications exhibit evident sequentiality but due to the contentions amongst other I/O submitting applications, the I/O accesses get multiplexed which leads to higher disk arm movements. BID schemes aim to exploit the inherent I/O sequentiality of Big Data applications to improve the overall I/O completion time by reducing the avoidable disk arm movements.

Unfortunately, despite its significance, the block layer, essentially the block I/O scheduler hasnt evolved to meet the volume and contention resolution needs of data centers experiencing Big Data workloads. We have designed and developed two Contention Avoidance Storage solutions in the Linux block layer, collectively known as *"BID: Bulk I/O Dispatch"* [Mishra et al. (2016); Mishra and Somani (2017)], specifically to suit multi-tenant, multi-tasking Big Data shared resource en-

vironments. In the this chapter, we discuss our first solution, i.e. a dynamically adaptable Block I/O scheduling scheme *BID-HDD*, for disk based storage. Chapter 4 discusses our second solution, *BID-Hybrid* for multi-tier storage deployments.

BID-HDD tries to recreate the sequentiality in I/O access in order to provide performance isolation to each I/O submitting process. Through trace driven simulation based experiments with cloud emulating MapReduce benchmarks, we show effectiveness of *BID-HDD* which results in 28% to 52% I/O time performance gain for all I/O requests than the best performing Linux disk schedulers.

In the next section, we briefly describe the working of the I/O stack, HDD characteristics and its inefficiencies in shared large data processing infrastructure

3.2 Background

In this section, we first briefly present the working of the Linux I/O stack in Section 3.2.1 followed by the discussion on the physical characteristics of Hard Disk Drives HDDs in Section 3.2.2. Section 3.2.3 and 3.3 discusses the I/O workload characteristics of Hadoop deployments and the requirements from a I/O scheduler in such environments, respectively. Section 3.4, describes the working of the current state-of-the-art Linux disk schedulers deployed in shared Big Data infrastructure.

3.2.1 Linux I/O Stack

The I/O stack of the data center architectures as shown in Figure 3.1, can fundamentally be broadly broken into *Applications, Host (OS) and Storage*. The difference between each of these solutions is in the layers of abstractions (storage virtualization) and the networking interconnects (Fibre Channel, RCoE, RDMA, etc.) between the storage and host [Bjørling et al. (2013); Islam et al. (2015); Bhadkamkar et al. (2009)]. Figure 3.2 is the simplistic representation of the Linux I/O stack [Bjørling et al. (2013); Malladi et al. (2016)]. In this section, we briefly present the working of the Linux I/O stack, focusing on the OS block layer. The block layer mediates and orchestrates

I/O requests from multiple applications to the underlying storage simultaneously. The following steps are taken to serve application's I/O request:

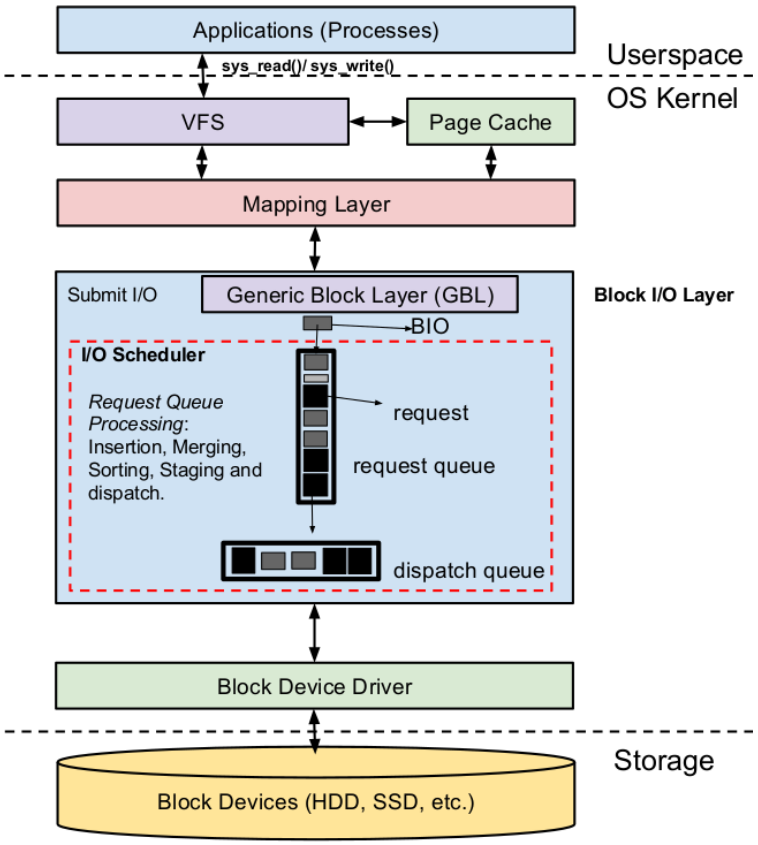


Figure 3.2: Architecture of Linux Kernel I/O Stack.

1. The Virtual File System VFS provides abstractions for applications (processes) to access storage devices via system calls. The calls include a file descriptor and the location [Bjørling et al. (2013); Xu et al. (2015); Avanzini (2014)]. VFS locates and determines the storage device as well as the file system hosting the data, starting from a relative location. VFS provides an uniform interface to access multiple file systems [Vangoor et al. (2017)].

2. While reading or writing from a file, the VFS checks if the data is present in the memory or page cache. If the data is not present, then a page fault occurs and the Mapping Layer is initiated to locate the data in the block device.

3. Kernel uses the "Mapping layer" to map the logical locations provided by the application (file descriptor) to the physical location in the respective block device. The Mapping Layer figures out the number of disk blocks required to be accessed. It should be noted that a file is stored in multiple blocks which may be distributed across multiple devices using logical volumes and on different devices may or may not be physically contiguous in the media. We assume that the logical volume on the physical media is sequential.

   The Mapping Layer and VFS enables storage virtualization functionalities such as logical volumes, heterogeneous storage pools or "tiers", etc.

4. After determining the physical locations of the blocks, the kernel uses the block layer to map I/O calls from the "Mapping layer" to the I/O operations (data-structures known as block I/O (**BIO**)).

   The I/O Scheduler in the block layer initializes the data structures called "requests", which represent I/O operations, to be sent to the device. I/O operations accessing non contiguous disk blocks (sectors) are broken into several I/O operations each accessing a contiguous set of blocks.

5. "Request" structures are then staged in a linked-list called *request queue*. The request queue allows I/O schedulers to sort, merge and coalesce the requests depending on the locations they access. Appendix A describes the relationships between the block I/O kernel data-structures used by the block layer to perform I/O operations.

6. Depending on the I/O Scheduling policies, "requests" scheduled to be sent to the device are dequeued from the "request queue" and enqueued to a structure known as "dispatch queue". I/O Scheduler maintains the dispatch queue and it's size is determined by the block device. Section 3.4 briefly discusses different schedulers currently employed in Linux block layer.

7. The "device driver" dequeues "requests" from the "dispatch queue" via service routines, which are then issued to the block device (HDDs, SSDs, etc.) using DMA (Direct Memory Access) operations.

The final interaction with the physical media is always in blocks (sectors in HDD, pages in SSD) and storage performance depends on the way storage is accessed. The block layer employs the *I/O Scheduler*, which provides the opportunity to coalesce requests and determines the order (& size) in which data is accessed from the block device. Therefore, the Block Layer is the most critical part of the I/O hierarchy.

The block layer for disk based storage (HDDs) has still remained highly volatile as the mechanical disks cannot support multiple hardware queues due to their physical constraints. Therefore, HDDs can have multiple software queues but single Hardware queue. The objective function of block layer for disk based storage is to optimize the request order from various applications in-order to recreate sequentiality of disk access and manage the I/O bandwidth for every application. BID schemes utilize multiple software queues in the block layer, but single hardware queue for delivering Software Defined Storage solutions for disk based storage devices.

3.2.2 HDD characteristics

Disk based storage devices (Hard Disk Drives HDDs) are the back-bone of data center storage. HDDs provide the perfect blend of cost and capacity as needed to accommodate the volume requirement of Big Data. The main research focus for a long time has been in improving physical media characteristics like increasing areal density of hard drives, read/write technology, etc. (for ex: shingled magnetic recording (SMR), heat-assisted magnetic recording (HAMR)) [Aghayev et al. (2017)].

The data in HDDs is organized as 512 byte (or 4KB emulated for newer drive technology) blocks in circular disk tracks and the data access time depends on both the rotational latency of disk platters and movement of read/write head mounted on disk arm. Therefore, sequential accesses (adjacent I/O blocks in the physical media) are fast as they depend on the rotation of disk platter (RPM of the disk) [Arpaci-Dusseau and Arpaci-Dusseau (2014)]. While random accesses are slow as they require the disk head to move from the current location to another track, i.e. involves disk arm movement which in turn is time consuming. Hence, the order in which the requests are sent

to the device is important. Therefore, the serving sequence of requests governs the overall I/O performance due to mechanical movement of disk arm.

The block layer I/O scheduler tries to sequentialize the requests to reduce both the number of seeks as well as the disk head traversal to the desired track. The time in processing the requests is important as they consume the I/O bandwidth of the device as well as increase the CPU wait times. This creates blocking (in the case of reads) in which the CPU waits for the data and doesn't issue more I/Os as well as doesn't do any meaningful work while waiting for the data [Bjørling et al. (2013); Bhadkamkar et al. (2009); Joo et al. (2017)].

3.2.3 Hadoop MapReduce: Working and Workload characteristics

Hadoop MapReduce [Dean and Ghemawat (2008); White (2012); Mishra et al. (2017)] is the de-facto large data processing framework for Big Data. Hadoop is a multi-tasking system which can process multiple data sets for multi-jobs in a multi-user environment at the same time [Islam et al. (2015); Ibrahim et al. (2011)]. Hadoop uses a block-structured file system, known as Hadoop Distributed File System (HDFS). HDFS splits the stored files into fixed size (generally 64 MB/ 128 MB) file system blocks, known as chunks, which are usually tri-replicated across the storage nodes for fault tolerance and performance [White (2012)].

Hadoop is designed in such a way that the processes access the data in chunks. When a process opens a file, it reads/writes in multiples of these chunks. Enterprise Hadoop workloads have highly skewed characteristics making the profiling tough with the "hot" data being really large [Harter et al. (2014)]. Thus, the effects of file system caching is negligible in HDFS [Harter et al. (2014); Krish et al. (2013)]. Most of the data access is done from the underlying disk (or solid state) based storage devices. Therefore, a single chunk causes multiple page faults, which eventually would result in creation and submission of thousands of I/O requests to the block layer for further processing before dispatching them to the physical storage.

Each MapReduce application consists of multiple processes submitting I/Os concurrently, possibly in different interleaving stages, i.e. Map, Shuffle and Reduce, each having skewed I/O re-

quirements [Ibrahim et al. (2011)]. Moreover, these applications run on multi-tenant infrastructure which is shared by a wide diaspora of such applications, each having different syntax and semantics. For Big Data multi-processing environments, although the requests from each concurrent process results into large number of sequential disk accesses, they face contention at the storage interface from other applications. These contentions are resolved by the OS Block Layer, more essentially the I/O scheduler. The inherent sequential operations of applications becomes non-sequential due to the working of the current disk I/O schedulers, which thereby result into unwanted phenomenons like multiplexing and interleaving of requests [Yi et al. (2017); Joo et al. (2017); Ibrahim et al. (2011); Krish et al. (2013)]. This also results in higher CPU wait/idle time as it has to wait for the data [Bjørling et al. (2013); Bhadkamkar et al. (2009); Nanavati et al. (2015); Joo et al. (2017)]. In order to provide performance isolation to each process as well as improve system performance, it is imperative to remove or avoid contentions.

Section 3.4 describes the working of the current state-of-the-art Linux disk schedulers deployed in shared Big Data infrastructure. In the next section, we discuss the requirements of a block I/O scheduler most suited for Hadoop deployments.

3.3 Requirements from block I/O scheduling in Big Data deployments

The key requirements from a block I/O scheduler in a multi-process shared Big Data environments, such as Hadoop MapReduce are as follows:

1. *Capitalize on large I/O access:* Data is accessed in large data chunks [White (2012)] (64/128 MB in HDFS), which have a high degree of sequentiality in the storage media. The I/O scheduler should be able to capitalize on large I/O access and should not break these large sequential requests.

2. *Adaptiveness:* Multiple CPUs (or applications) try to access the same storage media in a shared infrastructure, which causes skewed workload patterns [Kim et al. (2016)]. Additionally, each MapReduce task itself has varying & interleaving I/O characteristics in its Map,

Reduce and Shuffle phases [Ibrahim et al. (2011)]. Therefore it is imperative for an I/O scheduler to dynamically adapt to such skewed and changing I/O patterns.

3. *Performance Isolation:* In-order to meet the Service Level Agreements (SLAs), it is highly imperative to provide I/O performance isolation for each application [Kim et al. (2016); Park et al. (2016)]. For ex: A single MapReduce application consists of multiple of tasks, each consisting of multiple processes, each having different I/O requirements. Therefore, a I/O scheduler through process-level segregation should ensure I/O resource isolation to every I/O contending process.

4. *Regular I/O scheduler features:* Reducing CPU wait/idle time by serving blocking I/Os (reads) quickly; Avoid starvation of any requests; Improve sequentiality to reduce disk arm movements.

3.4 Issues with current I/O schedulers

Since version 2.6.33, Linux [Yi et al. (2017); Kim et al. (2016); Ibrahim et al. (2011)] currently employs 3 disk I/O Schedulers namely Noop, Deadline and Completely Fair Queuing CFQ.

As observed in Section 3.2.1, the main functionalities of the block I/O schedulers are as follows:

1. Lifecycle Management of the block I/O "requests" (which may consist of multiples of BIO structures) in the "request queue". Refer to Appendix A for details regarding the relationship of Block I/O data-structures.

2. Moving requests from "request queue" to the "dispatch queue". The dispatch queue is the sequence of requests ready to be sent to the block device driver.

The following example highlights the issues with the current Linux I/O Schedulers. For simplicity, we assume a Hard Disk Drive (HDD) with geometry of 1 platter, 100 sectors/track and 100 tracks/platter (see Figure 3.3). Consider 3 processes with process id's (**pid**) *A, B, C* submitting I/O requests to the disk block layer in the order shown in Table 3.1 (from top to bottom).

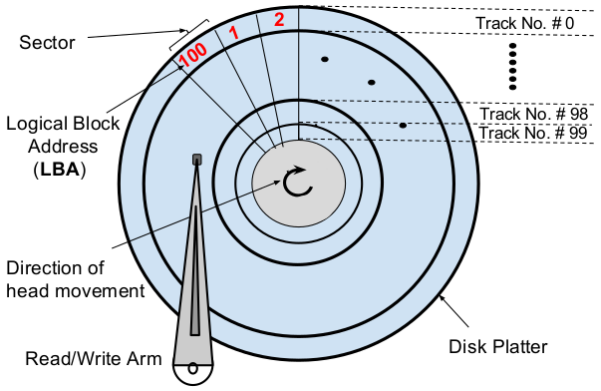


Figure 3.3: Geometry of the HDD with 1 platter, 100 sectors/track and 100 tracks/platter.

Table 3.1: I/O request Submission Order to the Block Layer.

order	**request**	**LBA**	**transfer size**	**Track No. (cylinder)**	**read/write**	**time to expire (ms)**
1	B1	7125	40	71	w	Exp#1
2	A1	305	24	3	r	Exp#2
3	A2	340	24	3	r	Exp#3
4	A3	370	24	3	r	Exp#4
5	C1	1600	4	16	r	Exp#5
6	B2	7165	40	71, 72	w	50
7	B3	7205	40	72	w	53
8	A4	410	24	4	r	60
9	A5	440	24	4	r	65
10	A6	470	24	4	r	100
11	C2	1670	4	16	r	105
12	B4	7245	40	72	w	110

In the table,

A_n, B_n, C_n**:** n^{th} request of processes A, B, C submitted to the "request queue;"

LBA: starting logical block address of the sorted "request" structure;

transfer size: number of disk blocks required for data transfer;

Track No.: the track (or tracks) where the entire request spans;

read/write: type of operation read 'r' or write 'w' performed by the request;

time to expire: time left in milliseconds at system time 'k' for the request to expire as per the deadline determined by the Deadline Scheduling Algorithm;

Exp#'x': Exp. denotes that the request has already expired and 'x' is the order

in which it has expired.

We assume that process *A and B* submit large I/O requests (transfer size) in short time intervals, while *C* submits small I/O requests in long time intervals.

The working of the three scheduling schemes of the current Linux block I/O Schedulers for this example are shown below:

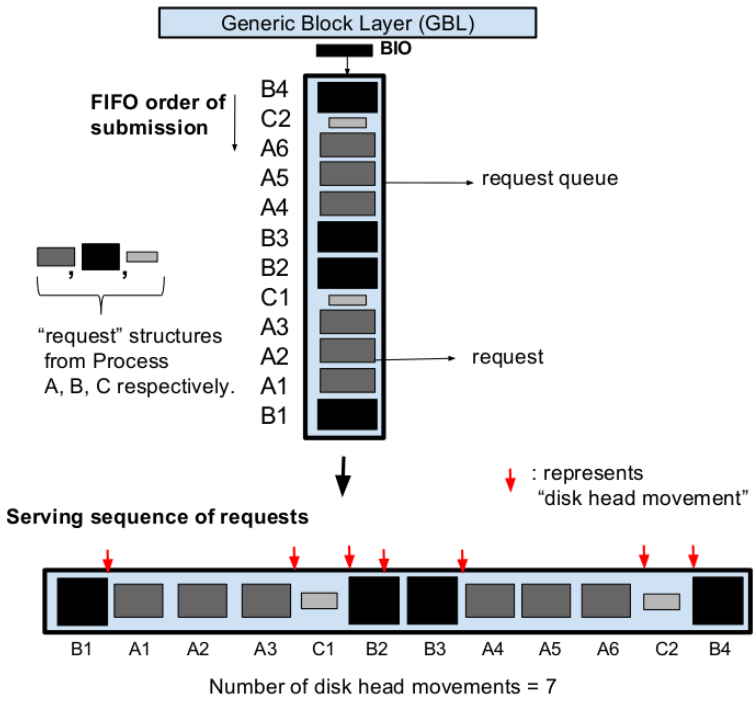


Figure 3.4: Working of *Noop Scheduling Algorithm.*

Noop: Noop is the simplest of the three scheduling algorithms. Figure 3.4 shows the scheduling order for the requests. As we see that its simply merges adjacent requests in queue, but does not perform any other operation (works on the principle of FIFO). The requests are served in the order in which they are submitted by the applications.

Observation: Noop is suitable for those environments where the number of processes submitting large I/O requests (*A and B*) concurrently is small. Noop can perform well in such a scenario where applications themselves submit large requests which have inherent sequentiality. For large

number of applications contending for the same storage media, Noop would cause large number of seeks due to multiplexing of requests from these processes. Adjacent requests (according to LBA) which arrive interleaved at the block layer (For ex: requests C1 and C2), are not provided the opportunity to coalesce and form sequences. Moreover, due to presence of requests from process C in between requests from bulky processes A and B, there is additional sequentiality loss of these bulky process. Also, if there are large number of processes like C (i.e. data transfer/seek is low) the disk I/O access time would increase significantly due to the FIFO nature of Noop.

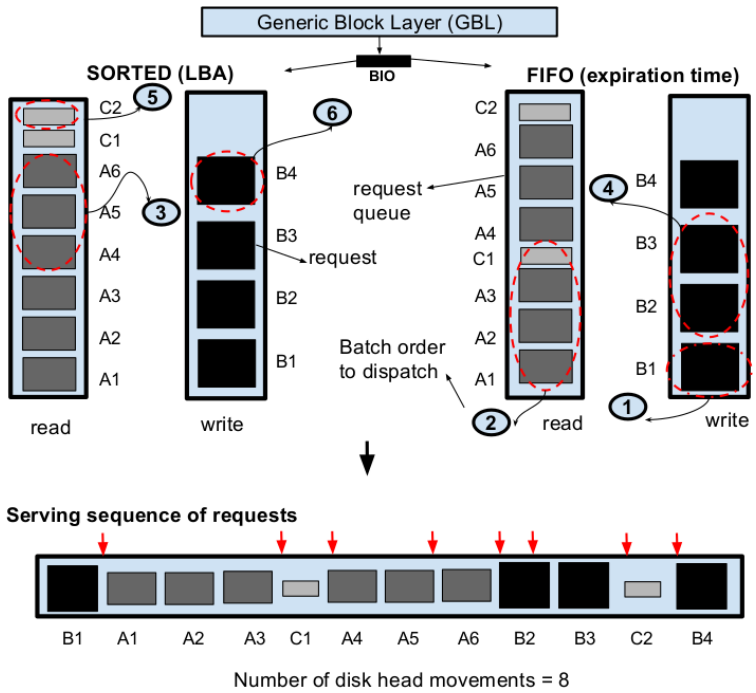


Figure 3.5: Working of *Deadline Scheduling Algorithm.*

Deadline Scheduler: The Deadline Scheduler tries to prevent starvation of requests. Each request is assigned an expiration time (reads=500ms, writes=5000ms) [Yi et al. (2017); Ibrahim et al. (2011)]. There are two kinds of queues: *Sorted Queues*, where requests are sorted by disk access location and *FIFO Queues*, where requests are ordered according to deadline [Kim et al. (2016); Schnberger (2015)]. Some implementation have just three queues: 2 FIFO queues and a common Sorted Queue [Yi et al. (2017)]. For simplicity, we consider the former implementation

with both FIFO and Sorted queues having separate Read and Write queues as shown in Figure 3.5. The requests in the sorted queues are processed in batches (*fifo_batch*). The deadline scheduler keeps issuing request batches to the dispatch queue from the sorted queues unless the request at the head of the Read/Write FIFO queue expires [Yi et al. (2017); Inc. (2015); lin ()].

Deadline Scheduler, despite its name, does not provide strict deadlines and actual I/O waiting times can be much higher. The selection of batches of requests from the queues is based on expiry of requests, otherwise requests are served from the sorted queues.

For the given example, we consider at system time 'k' the **time to expire**, i.e., the time left for expiration of each request, as determined by the Deadline Scheduling Algorithm. Deadline Scheduler tries to first dispatch those requests whose deadlines have already expired.

The "requests" from all the processes are staged in sorted (according to LBA) and FIFO (according to time_to_expire), in respective read and write queues, as shown in Figure 3.5.

The selection of batches in which the requests are served as per Deadline Scheduling scheme is as follows:

$batch_1 : \{B1\} \rightarrow$ **writeFIFO**;

$batch_2 : \{A1, A2, A3, C1\} \rightarrow$ **readFIFO**;

$batch_3 : \{A4, A5, A6\} \rightarrow$ **readSORTED**;

$batch_4 : \{B2, B3\} \rightarrow$ **writeFIFO**;

$batch_5 : \{C2\} \rightarrow$ **readSORTED**;

$batch_6 : \{B4\} \rightarrow$ **writeSORTED**.

Hence, $batch_j$ is the selection order of dispatch of a batch of request. Also, the arrow "$\rightarrow$" points to the I/O queue from which the batch is selected.

Here we see that $batch_1$ has only 1 request 'B1' as its expiration is earlier than any other request in the write FIFO queue as well as requests ($batch_2$) in the read FIFO queue have already expired. Once the expired requests are served, the scheduler picks batches from sorted queues ($batch_3$). While serving all these requests, B2 & B3 expire, hence they are scheduled ($batch_4$). We observe that $batch_5$ and $batch_6$ also contain just one request C2 and B4, respectively. This is due to all the

requests already been scheduled from their respective batches. The switching of batches causes high number of disk seeks. Moreover, when multiple processes of the same type submit I/O requests at the same time, this also adds to increased latency.

Observation: For processes (such as *A and B*), which submit large I/O requests in short time intervals, deadlines of the requests would expire at the same time. The FIFO queues would have large number of requests whose deadlines have expired. Moreover, smaller processes such as *C*, might still suffer from long waiting time because of a large number of pending requests from other processes. With multiple processes submitting requests at the same time and expiration time being close, deadline scheduler would cause deceptive idleness [Seelam et al. (2005)]. Deceptive idleness is a condition when the scheduler would select requests from processes, leading to increased disk head seeks to disjoint locations in the disk. Thereby, Deadline based I/O scheduling leads to reduced throughput and result in large number of seeks [Yi et al. (2017); Kim et al. (2016)] for highly sequential and multi-process workloads like Hadoop MapReduce.

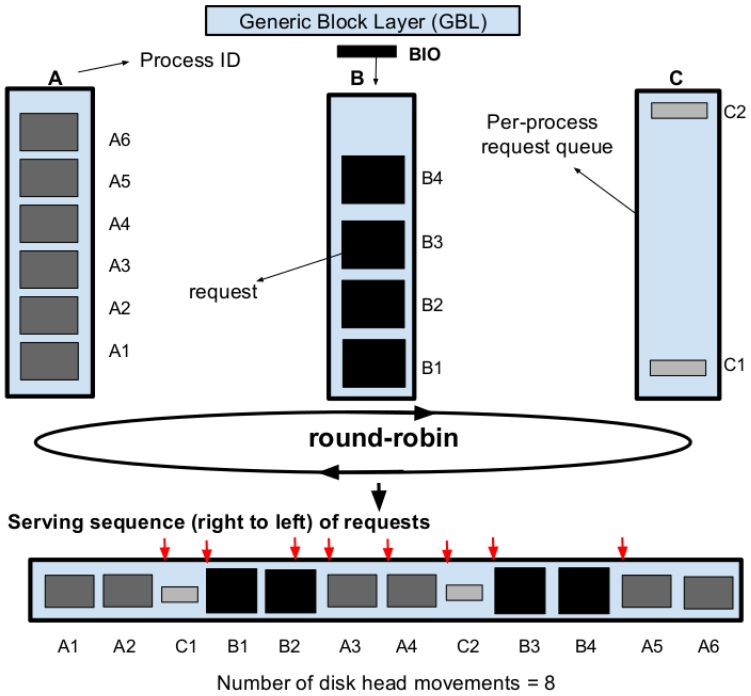


Figure 3.6: Working of *Completely Fair Queuing* (CFQ)

Completely Fair Queuing (CFQ): CFQ is the default disk I/O scheduler in the current Linux distribution [Kim et al. (2016); Park et al. (2016)]. It divides the available I/O bandwidth among all the contending I/O request submitting processes [Yi et al. (2017)]. CFQ maintains a location sorted queue for every process for synchronous (blocking) I/O requests and batches together asynchronous (non-blocking) requests from all processes in a single queue. During its time slice, a process submits requests to the dispatch queue which is governed by setting the parameter **quantum** [Inc. (2015)]. CFQ is suitable for environments where all processes need equal and periodic share of the block device like interactive applications.

In Figure 3.6, we see that CFQ maintains per-process queues and requests from each process (For ex: *A, B, C*).

The requests in the per-process request queue RQ_{pid}, where pid is the process id, are as follows:
$RQ_A : \{A1, A2, A3, A4, A5, A6\}$;
$RQ_B : \{B1, B2, B3, B4\}$;
$RQ_C : \{C1, C2\}$;

CFQ inserts requests to the dispatch queue in a round robin fashion according to "quantum," which are then sorted in the dispatch queue. Thereby, in the first cycle (A1, A2), (B1, B2) and (C1) are selected in round-robin from each process request queue RQ_A, RQ_B, and, RQ_C, respectively. Similarly {(A3, A4), (B3, B4), (C2)} & {(A5, A6)} in the second and third cycle, respectively. In the "dispatch queue", the requests are sorted according to their logical block address (LBA) values.

The final order of requests being served using CFQ is as follows:
{A1, A2, C1, B1, B2, A3, A4, C2, B3, B4, A5, A6}

Observation: From Figure 3.6, we observe that due to round-robin fashion of selection of requests, the disk head movement follows the access pattern (accessing the same regions of the disk in a cyclic pattern). CFQ in its quest of being fair to all processes, resulting into disk head movements leads to a higher latency and increased queue depth. One solution would be to increase the number of

requests dispatched from a process queue, but this would lead to long latency for systems with a large number of processes. Processes like *A and B* would consume a large portion of the disk I/O time due to their large data access requirements. CFQ is biased towards synchronous processes (with each having their own process queue) and all other asynchronous processes in one queue. However, application with large data access requirements and skewed workloads like MapReduce would suffer high latency due to their specific and disjoint disk seeks. CFQ is undesirable for a multi-process environment with diverse disk I/O characteristics within request queue contending processes [Yi et al. (2017); Kim et al. (2016); Park et al. (2016)].

A fourth I/O Scheduling scheme, Anticipatory Scheduler has been discontinued from the Linux kernel. It associates a fixed waiting time (6ms) for every synchronous (read) request [Yi et al. (2017); Kim et al. (2016); Ibrahim et al. (2011)]. In MapReduce environments, this would lead to increased CPU waiting time as well as lead to starvation of large number of requests.

Takeaway: In summary, due to contention amongst different processes submitting I/O to the storage device and the working of the current I/O schedulers, the inherent sequentiality of MapReduce processes are lost. They result into unwanted phenomenons such as interleavings and multiplexing [Joo et al. (2017)] of requests sent to the device, thereby also adversely affecting system performance (CPU wait time, etc) and increasing latency in disk based (HDDs) storage systems. We observe that the existing Block I/O schedulers do not support the set of requirements laid down in Section 3.3 and there is a clear need of new I/O scheduling scheme for such Big Data deployments.

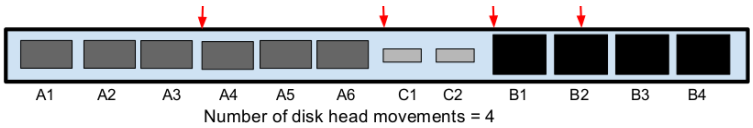


Figure 3.7: Working of *an Ideal Scheduling Algorithm.*

Figure 3.7 shows a sequence of requests that would be dispatched by an "Ideal" scheduler suitable for MapReduce type applications. We notice, that this scheduler has minimal disk head movements as well as provides high throughput (maximizing sequentiality). Such a scheduling

scheme needs to be intelligent and dynamically adaptable to changing I/O patterns. Further, such a scheduler should take into consideration all the requirements laid out earlier in this section.

3.5 BID-HDD: Contention Avoiding I/O Scheduling for HDDs

HDDs form the backbone of data centers storage. The effects of caching is negligible in an enterprise Big Data environment [Harter et al. (2014); Krish et al. (2013)] (refer to Section 3.2.3), therefore large number of page faults occur, which in turn result in most of the data accesses from the underlying storage. Hence, it is imperative to tune the data management software stack to harness the complete potential of the physical media in highly skewed and multiplexing Big Data deployments. As discussed in earlier sections, the block layer is the most performance critical component to resolve disk I/O contentions along the odyssey of I/O path. Unfortunately, despite its significance in orchestrating the I/O requests, the block layer essentially the *I/O Scheduler* has not evolved much to meet the needs of Big Data.

We have designed and developed "BID-HDD: Bulk I/O Dispatch for Hard Disk Drive" in the Linux block layer specifically to suit multi-tenant, multi-tasking shared Big Data environments. Essentially, we develop a Block I/O scheduling scheme *BID-HDD* for disk based storage to manage I/O contentions. BID-HDD tries to recreate the sequentiality in I/O access in order to provide performance isolation to each I/O submitting process.

BID-HDD is designed taking into consideration the requirements laid out earlier in Section 3.3. BID as a whole is aimed to avoid contentions for storage I/Os following system constraints without compromising the SLAs.

BID-HDD aims to avoid multiplexing of I/O requests from different processes running concurrently. To achieve this, we segregate the I/O requests from each process into containers. The idea is to introduce dynamically adaptable and need-based anticipation time for each process, i.e. time to wait for adjoining I/O request. This allows coalescing of the bulky data accesses and avoid starvation of any requests. Each process container has a wait timer, based on inter-arrival time of requests and deadline associated with it. The expiry of either marks the container to be flushed in

order to the storage device. This forms a pipeline of large data blocks from adjoining locations in the disk.

In order to achieve the above, we modify the existing Host Block Layer by using the following queues:

request queue RQ: Whenever a block I/O "request" is submitted by an application it is enqueued in the *request queue*. Similar to the existing I/O schedulers, BID-HDD uses the request queue to: 1) coalesce (merge) the requests accessing adjoint LBAs; 2) split the requests accessing non-contiguous disk locations into multiple requests, each accessing contiguous locations.

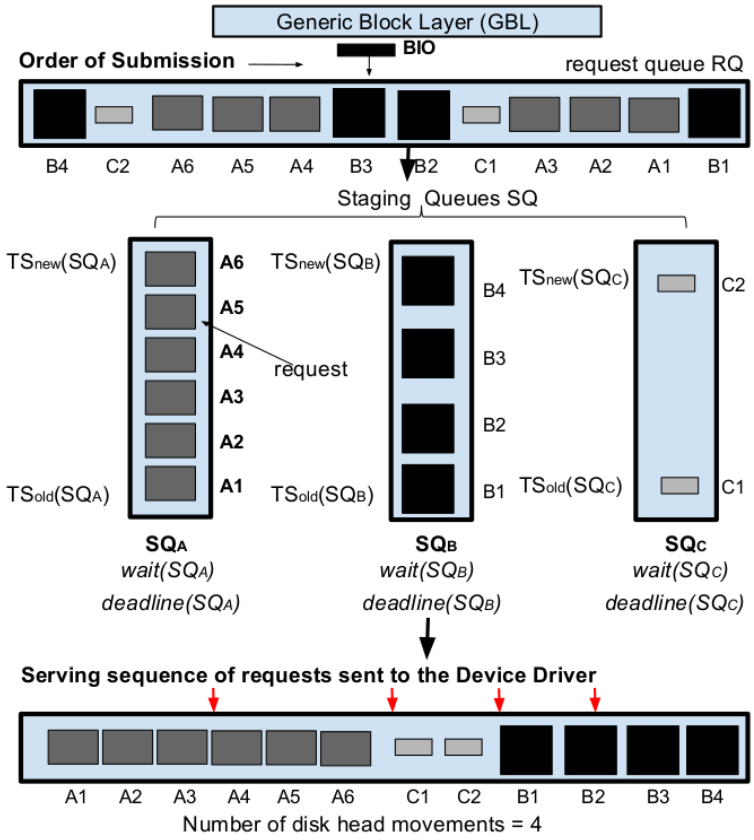


Figure 3.8: Working of *BID-HDD*

Per process staging queues SQ_p***:*** In order to segregate the I/O requests from each process, BID uses separate containers known as *staging queues* for each process. BID-HDD groups the I/O requests into staging queues on the basis of the process id (**pid**) they belong to. The staging queue

for a process p is denoted by SQ_p. SQ_p's are not permanent queues and are only created whenever the I/O requests of process p present in RQ are ready to be staged and there is no existing SQ_p. The staging queue for a process holds the requests which are ready to be sent to the dispatch queue of the device (based on block device driver specifications.) The staging queue is important multi-fold: 1) for segregating I/O requests from each process; 2) provide more coalescing opportunities under the assumption that bulky processes, send a large number of requests to adjoining locations in the physical media (For ex: 64 MB HDFS blocks;) 3) provides BID dynamic adaptability to changing workload patterns. This is achieved through the following parameter associated with each staging queue SQ_p.

- Time stamp of the oldest request present in the queue, denoted by $TS_{old}(SQ_p)$.
- Time stamp of the newest request present in the queue, denoted by $TS_{new}(SQ_p)$.
- Wait timer for next I/O request $wait(SQ_p)$.
- Flush deadline timer $deadline(SQ_p)$.

dispatch queue DQ: The *dispatch queue DQ* holds the requests which are ready to be sent to the block device. The order of requests sent to the *dispatch queue* is managed by the *I/O Scheduler*, while the device driver specifications decide the number of requests the dispatch queue can hold at a time. The requests inside the dispatch queue are sorted according to logical block addressing LBAs. The requests from the *dispatch queue* are dequeued according to the disk controller on the physical device.

Figure 3.8 shows the working of BID-HDD with the help of the I/O submission order as in Table 3.1. We now describe the working of BID[1] in terms of the path the I/O requests follow from the generic block layer to the device driver:

Enqueuing I/O request in request queue (RQ): The block layer synchronizes the access to shared exclusive resource, i.e. the *request queue*. The lock needs to be acquired by the process which

[1]BID and BID-HDD is used interchangeably throughout the chapter as BID-Hybrid also uses BID-HDD.

ALGORITHM 1: Stage Requests

```
for every process p ∈ P do
    if SQ_p not present then
        Create SQ_p;
        Create and set wait(SQ_p) and deadline(SQ_p) with default values;
    if SQ_p not marked for flushing then
        Dequeue R_p from RQ and enqueue in SQ_p;
        Reset wait timer wait(SQ_p);
        if R_p contains a blocking I/O request then
            if Remaining time in deadline(SQ_p) > 500ms. then
                Reset deadline timer deadline(SQ_p) = 500ms;
```

inserts the block I/O request structures to the *request queue* Bjørling et al. (2013). Enqueuing in the request queue depends on the free space of the "request queue" and a block I/O request can only be inserted if the request queue RQ is not full. If the block I/O request can be merged with any existing requests, it is merged otherwise it forms a separate request structure.

Dequeuing I/O request from request queue (RQ) to staging queues (SQ): Let R denote the set of requests currently present in "request queue". Let P denote the set of processes which have their requests currently enqueued in request queue. Let R_p denote the set of I/O requests out of R which belong to process $p \in P$. The I/O requests are dequeued from request queue and enqueued in the corresponding staging queue as described in Algorithm 1.

Wait timer for Staging queues: As discussed in Sections 3.2.3 & 3.3, to ensure efficient resource utilization as well as performance isolation of every I/O contending process, it is critical that the scheduler is dynamically adaptable to changing and skewed I/O patterns. BID gets its dynamic adaptable capability by introducing *per staging queue wait timer* "$wait(SQ_p)$". The wait timer $wait(SQ_p)$ value for a staging queue SQ_p is determined as follows: Whenever a set of requests R_p is enqueued to SQ_p, the difference between the timestamp of newest request present in the SQ_p denoted by $TS_{new}(SQ_p)$ and the time stamp of the oldest I/O request present in set R_p is computed. BID remembers k most recent time difference values and uses their weighted mean as the wait timer value. Whenever SQ_p is created, i.e. when the historic k time difference values are not available, the value of wait timers is set to a default value.

ALGORITHM 2: Flush Requests: Pipelining

for *every "staging queue" marked for flushing, select* SQ_p *which was marked earliest.* **do**
Dispatch all I/O requests from SQ_p to DQ;
Delete SQ_p;
Delete $wait(SQ_p)$ and $deadline(SQ_p)$;

The main idea here is to exploit the inter-arrival time of batches of requests from a process to profile the processes I/O characteristics. The wait timer, therefore provides more opportunity to coalesce adjoining requests from a process for maintaining sequentiality as well as in the same time avoid multiplexing from other processes.

It can be seen that the wait timer $wait(SQ_p)$ is dynamic and adapts to the changing process I/O characteristics. However, the wait timer is also deleted along with SQ_p after flushing. Whenever the wait timer $wait(SQ_p)$ for SQ_p is expired, SQ_p is marked for flushing.

Deadline timer for staging queue: Use of wait timer alone can cause starvation, as staging queue SQ_p which always gets enqueued with request(s) before $wait(SQ_p)$ expires will never be flushed. Additionally, a non-bulky process might suffer due to large wait time. To avoid such situations BID employs a deadline timer. The deadline timer $deadline(SQ_p)$ of a staging queue SQ_p indicates maximum allowable time the queue SQ_p can exists before marked for flushing. The deadline of a staged queue SQ_p depends on the type of requests in the staging queue SQ_p. If there are only non-blocking I/Os (writes) in SQ_p, $deadline(SQ_p)$ is set initially to $5000ms$. Whenever a blocking I/O (read) request is enqueued to SQ_p, the $deadline(SQ_p)$ is set to $500ms$ if its current value is more than $500ms$. The reseting of deadline $deadline(SQ_p)$ ensures that blocking I/Os do not encounter higher delays. Whenever $deadline(SQ_p)$ expires, SQ_p is marked for flushing. The deadline timer also ensures that a process with high disk I/O (bulky) does not starve another process with lighter disk I/O (non-bulky.)

Marking staging queue for Flushing: BID-HDD marks a staging queue for flushing whenever any of the timers ($wait(SQ_p)$ or $deadline(SQ_p)$) expires. Flushing denotes the process of sending

the I/O requests currently enqueued in SQ_p to the dispatch queue (DQ.) BID keeps track of the order in which the staging queues are marked for flushing.

Flushing I/O requests from staging queues to dispatch queue: BID-HDD dequeues the I/O requests from staging queues and enqueues them to dispatch queue. As discussed in Algorithm 2, BID dispatches the requests from the earliest marked staging queue and follows the marking sequence. The size of dispatch queue depends on the device driver specification and all the I/O requests from staging queue may not get dispatched at once. BID ensures that a staging queue is fully flushed before considering the next marked staging queue. This prevents multiplexing of I/O requests, thereby involves less movement of the disk arm to disjoint locations in the physical media.

In BID-HDD, the efficient pipelining of large data blocks groups (as shown in Figure 3.8) from adjoining locations in the disk leads to reduction in disk arm movements (leveraging sequentiality performance) along with dynamic and need-based anticipation time ensures performance isolation to each I/O contending processing following system constraints without compromising the SLAs. BID-HDD is essentially a contention avoidance technique which can be modeled to cater different objective functions (storage media type, performance characteristics, etc.).

Using the above for contention avoidance storage solution, BID-HDD is capable of delivering higher performance. In the next section, through trace-driven simulation experiments using cloud emulating Hadoop benchmarks, the performance of BID-HDD is evaluated and compared with the current Linux scheduling schemes.

3.6 Experiments and Performance Evaluation

Through trace-driven simulations and in-house developed system simulators, we conducted experiments for evaluating the performance of our scheme, i.e. *BID-HDD*.

3.6.1 Testbed: Emulating Cloud Hadoop workloads and capturing block layer activities.

For our experiments, we select industry and academia wide used Hadoop benchmarks considering a wide diaspora of I/O workload characteristics, as specified in HiBench [Huang et al. (2010)] & TPC Express Benchmark (TPCx-HS)- Hadoop suite [TPC(tm) (2016)]. These benchmarks have been designed to recreate enterprise Hadoop cloud environments, stressing the hardware and software resources (storage, network and compute) as observed in production environment. For example, TeraSort is a popular compute and disk intensive MapReduce benchmark used for emulating cloud environment workloads under heavy load with multiple chained MapReduce processes running concurrently. Consider Table 3.2 for the set of Hadoop workloads with varying I/O characteristics we used for the capturing the block I/O layer activities.

Our experimental testbed, see Figure 3.9, consist of our Hadoop cluster and Trace collection nodes. We ran the benchmarks on our Hadoop cluster having Hadoop v2.6.5 with latest implementation of YARN resource negotiator. The cluster topology consists of one NameNode and 8 DataNodes, each with two 4-core AMD Operon 2354 processor, 8 GB Memory & 250 GB Serial ATA (SATA) HDD.

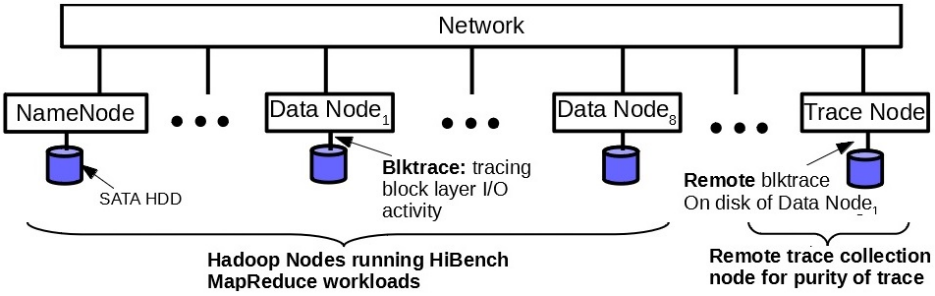


Figure 3.9: Experimental Testbed: Hadoop cluster & capturing block layer I/O activity using **blktrace**.

We collect traces from the block layer of a disk in a DataNode in such a stage where the applications have submitted block I/O structures to the block layer using the *blktrace* [Brunelle (2007)] linux utility. Blktrace aids to captures the complete block layer I/O activities of a block

device, right from I/O submission by process to completion of the request from the device. The traces at this stage is important for our simulation based experiments to emulate the functioning of the block layer before submission to the I/O scheduler. The traces include details such as process id (**pid**), CPU core submitting I/O, logical block address (LBA), size (no. of 512 byte disk blocks), data direction (read/write) information for each I/O request. Please note, we collected (stored) the traces remotely on a different machine through the network and not stored in the same local HDFS disk for maintaining the purity of the traces & minimize the effects of the SCSI bus [Brunelle (2007); Riska et al. (2007); Chen et al. (2011)].

Table 3.2: Cloud Emulating Hadoop Benchmarks: I/O characteristics.

Workload	**I/O Characteristics**
Grep	Mostly sequential reads with small writes.
Random Text Writer	Mostly sequential writes, mixed with random writes and negligible reads.
Sort	More reads than writes. Large sequential reads with random writes and later sequential writes.
TeraSort	Good mix of sequential and random reads/writes. More reads than writes.
Wordcount	Mostly sequential reads, with large number of random writes followed by random reads and small sequential writes.
Word Standard Deviation	Mostly sequential reads with small inter-phase writes, followed by small writes in the end.

3.6.2 System Simulator

We have designed and developed a System Simulator using Python v2.7.3 to replicate the working of the System level components (Host OS, Storage devices, etc.). We use the trace file (as discussed in Section 3.6.1) for application I/O submission order. The Simulator has two major modules: a) *OS Simulator:* Takes the order of I/O submissions and performs Linux Kernel block layer functions (contains pluggable I/O Scheduler sub-module); and b) *HDD Simulator:* Takes input from OS Simulator and returns performance metrics. The details of each of the components (see Figure 3.10) is discussed below.

- **OS Simulator:** This module takes the collected workload I/O traces (Trace File) as input and recreates the Kernel Block Layer functions after the stage from which the traces were

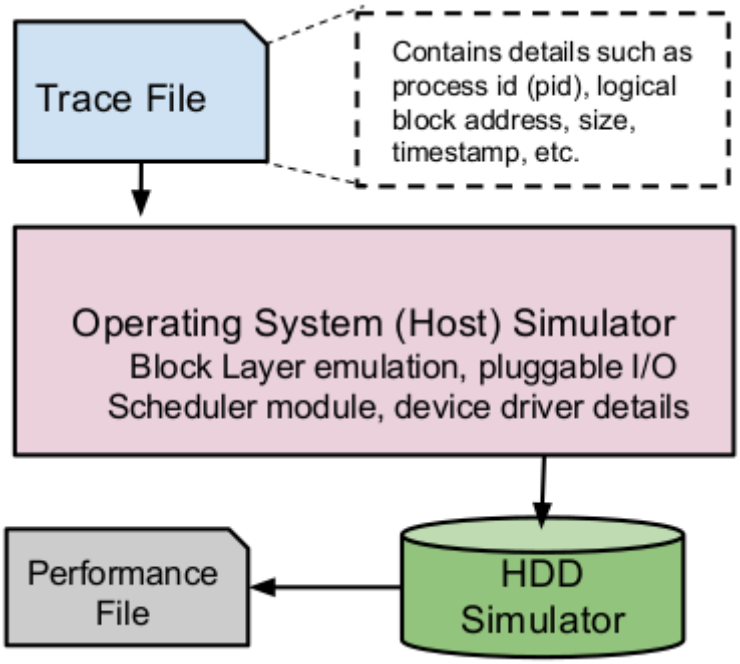


Figure 3.10: Simulator Components.

collected (refer to Section 3.6.1). It performs Kernel block I/O operations such as: 1) making block I/O (**BIO**) structure from traces; 2) Enqueuing BIO request structures to the "request queue RQ" based on RQ limitations; 3) Pluggable I/O Scheduling: merging, sorting, re-ordering, staging, etc. as per the Scheduling scheme; 4) managing the I/O requests inflow and outflow in the "dispatch queue" as per the device driver specifications; 5) dispatching requests from dispatch queue to the block device. The I/O Scheduling sub-module is made pluggable so that different scheduling schemes can be tested. Simulator provides the flexibility to configure parameters like: data holding size of each BIO structure, request queue size, dispatch queue size and block device driver parameters. To preserve the I/O characteristics of the workloads, the requests are submitted based on the timestamp from the trace file to the kernel block layer.

- **HDD Simulator:** This module takes the I/O requests from the dispatch queue of the OS Block Simulator and based on the device type (HDD), return performance metrics like

completion time depending on the current state of the block device. The module takes block device configuration parameters as inputs (device driver) such as drive capacity, block device type (HDD), etc. For HDDs, drive parameters include geometry, no. of disk heads, no. of tracks (cylinders), sectors/track, rotations per minute (RPM), command processing time, settle time, average seek time, rotational latency, cylinder switch time, track-to-adjacent switch time, and head switch time. The HDD Simulator is CHS compliant for 48-bit logical block addressing (LBA). The HDD simulator calculates the I/O access time (per I/O request) by HDDs considering the current location of the disk arm and time needed to reach the desired new location and access data size. The access time also takes into account minute details such as command processing time, settle time, rotational latency, cylinder (track) switch time, head switch time and average seek time [Bian et al. (2017); Ruemmler and Wilkes (1994); Arpaci-Dusseau and Arpaci-Dusseau (2014)]. The configurable features gives us the ability to test the schemes with different devices as well as drive architectures.

3.6.3 Performance Evaluation: Results and Discussions

We compare the effectiveness of our Contention Avoidance or block I/O Scheduling scheme, BID-HDD, with the two best performing Linux kernel block I/O schedulers used in the enterprise deployments, namely, CFQ and Noop. CFQ performs well in almost all workloads in terms of I/O bandwidth fairness, while Noop is selected due to its superior performance in some MapReduce workloads which have high degree of sequentiality [Yi et al. (2017); Ibrahim et al. (2011)]. Deadline I/O Scheduling leads to reduced throughput and result in large number of seeks [Yi et al. (2017); Kim et al. (2016)] for highly sequential and multi-process workloads like Hadoop MapReduce. As the processes submit large number of I/Os in short interval of time, therefore, this leads to expiry of most of the requests in the queue and it eventually acts as a FIFO queue (refer to Section 3.4). Hence, we compare our solutions with CFQ and Noop.

For our experiments, we use the default parameters as shown in Table 3.3, which is based on the storage devices and driver specifications.

Table 3.3: Block Device Parameters in use for Performance Evaluation.

Block Device	Default Parameters

SATA HDD	maximum "request" structure size = 512 KB;
	request queue size = 256 BIO structures (128 reads, 128 writes);
	max. size of each block I/O (BIO) structure = 128 x 4K pages;
	1 page (bio vec) = 8 x 512-byte disk sectors (block);
	access granularity (disk block sector size) = 512 bytes.
	Specification based exactly as our 250 GB Hadoop cluster HDD.

Based on trace-driven simulations, we analyze the performance of different block level contention avoidance schemes, i.e. BID-HDD, CFQ, and Noop.

3.6.3.1 Cumulative I/O Completion Time

Figure 3.11 represents the cumulative time taken (x-axis) by the block device to fulfill all the I/O requests[2] using different schemes. This graph shows the effectiveness of the scheduling schemes, as the order in which the I/O requests are submitted to a block device plays a significant time in deciding the time taken to fulfill them.

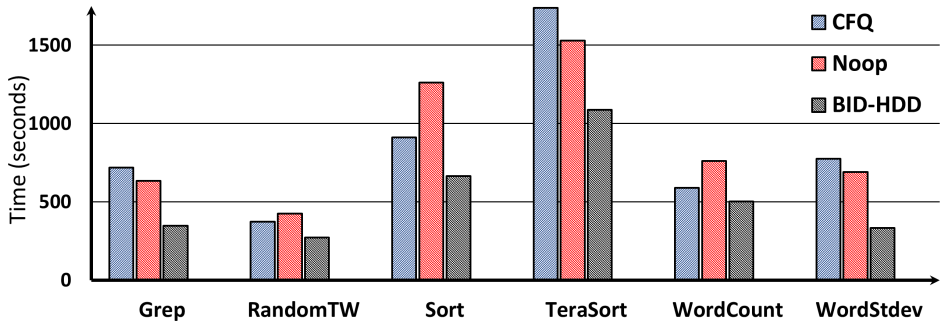


Figure 3.11: Cumulative I/O Completion Time.

Figure 3.11 demonstrates that BID-HDD outperforms CFQ and Noop for all the workloads. The savings in cumulative I/O completion time is maximum for *WordStandardDeviation* & *Grep*, which have a relatively higher degree of sequentiality than others. BID-HDD requires only about 50% of the time taken by CFQ to serve the same set of I/O requests.

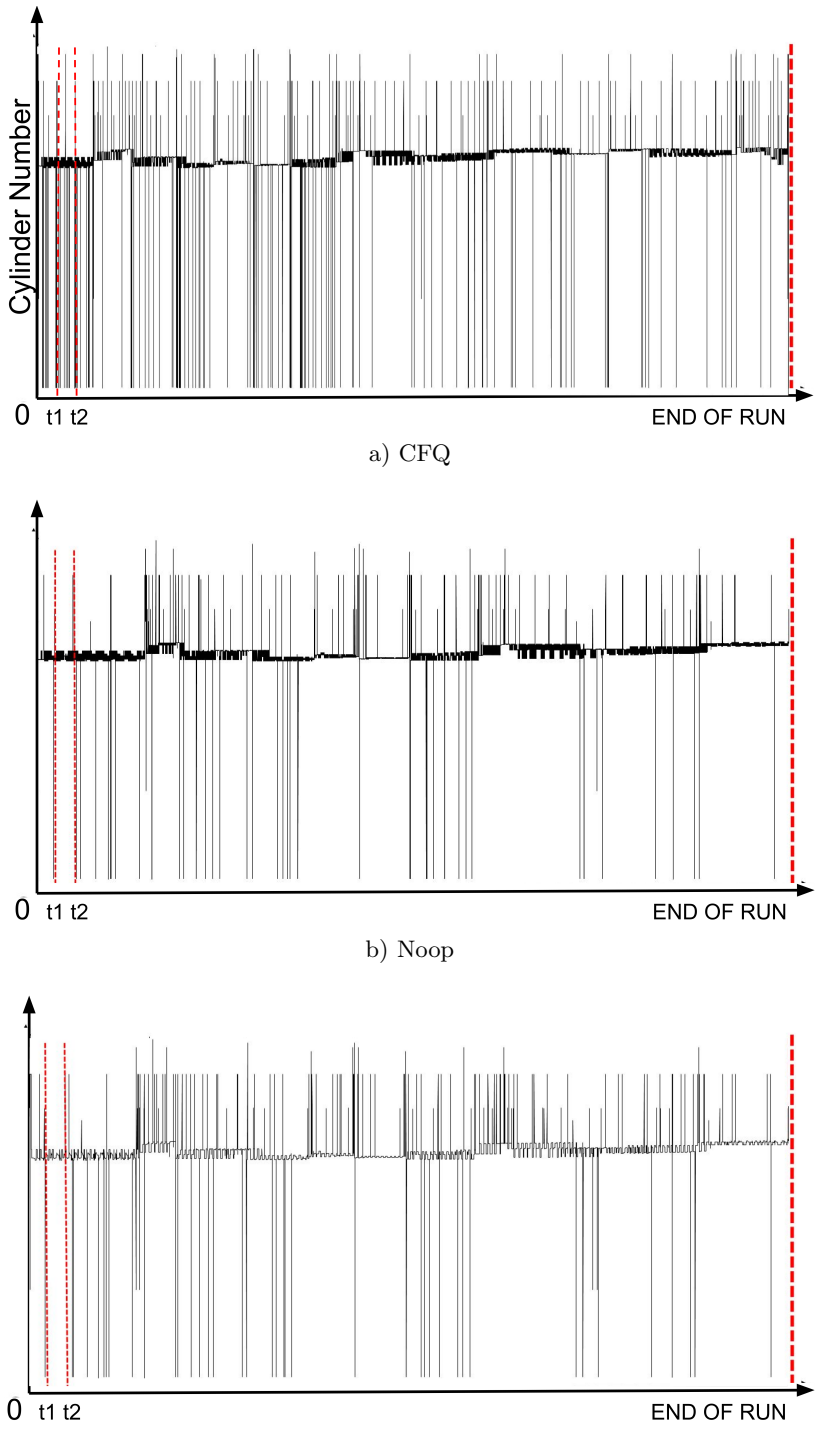
[2] An I/O request can access data sectors located on adjoining disk cylinders (tracks).

An interesting observation is that Noop outperforms CFQ, requiring 12% lesser time for workloads with higher inherent sequentiality in I/O accesses. The FIFO characteristics of Noop, tends to preserve the sequentiality of processes, whereas CFQ in the advent of being fair to all contending processes (in terms of I/O bandwidth), multiplexes the requests. This nature of CFQ is evident from Figure 3.12, which shows the disk arm movements in terms of HDD track accesses (y-axis) during the course of *WordStandardDeviation* workload. CFQ results in higher number of disk arm movements between tracks (more vertical lines), thereby resulting in higher I/O completion time due to round-robin switching of per-process queue.

Figure 3.12b & 3.12c, show very similar track or I/O access pattern in Noop and BID-HDD, respectively, yet there is a significant difference in the cumulative I/O access times. A careful examination reveals that though the number of long distance track changes could be similar, the number of short distance track changes (density of black lines) are much larger in Noop than in BID-HDD. From Figure 3.12a and 3.12c, it is observed that BID reduces both the long strokes as well as the short strokes as compared to CFQ. Due to staging capabilities and dynamic adaptability, BID-HDD makes justified decisions, thereby reducing the number of head movements as well as increasing the opportunity to coalesce requests together. This is evident from Figure 3.13, which shows the magnified view of Figure 3.12b, 3.12a and 3.12c between timestamps t_1 *and* t_2.

We believe there is some kind of *Amortization effect* occurring due to bulkiness of I/Os. We notice from Figure 3.13, that Noop has rigorous disk head movements[3], while BID-HDD linearizes depicting the serving of I/Os in bulk to storage and reducing preventable disk arm movements. Few initial I/Os of the sequential group might experience a higher latency, however, due to lower latencies experienced by the later I/Os, their overall average latency is reduced. We also observe the dynamic adaptable capability of BID-HDD especially in skewed workload environments. For every process the I/O bandwidth time changes depending on the workload characteristics and process I/O profiling.

[3]We use the terms "Disk Arm Movements" and "Disk Head Movements" interchangeably in this document.



a) CFQ

b) Noop

c) BID-HDD

Figure 3.12: Disk arm movements for *WordStandardDeviation* workload.

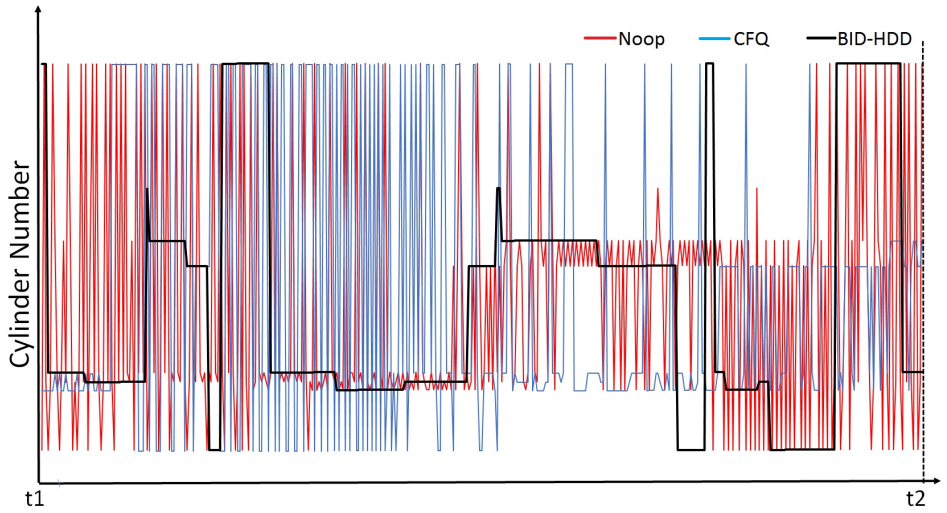


Figure 3.13: Disk head movements for Noop, CFQ and BID-HDD between timestamps t_1 and t_2 for *WordStandardDeviation*.

Takeaway 1: BID-HDD handles the contention at the block layer while preserving the inherent sequentiality (bulkiness) of processes in all MapReduce workloads. This results in fewer disk arm movements, leading to reduction in I/O access time as compared to other scheduling schemes.

Takeaway 2: Noop can result in better I/O performance than CFQ for highly sequential workloads, as Noop can maintains the sequential order but CFQ in the effort of being fair to all processes leads to more disk seeks thereby higher I/O completion time.

3.6.3.2 Number of Disk Arm Movements

Figure 3.14 shows the disk arm movements incurred by all workloads. BID-HDD and BID-Hybrid leads to fewer disk head movements as compared to all other scheduling schemes in all the MapReduce workloads. The amortization affect of BID attributes to the reduction in disk arm movements, as discussed in Section 3.6.3.1 and Figure 3.13. In BID-HDD, the dynamically adaptable anticipation and the efficient pipelining of flushed requests from staging queues of processes are responsible for capitalizing the sequentiality, thereby reducing the disk arm movements. Workloads which have a high degree of sequentiality experience the maximum reduction in entropy of disk arm movements.

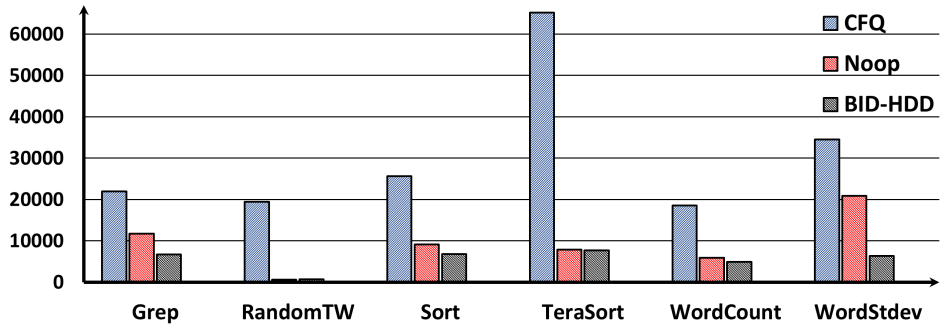


Figure 3.14: Total number of disk arm movements.

Please note that Figure 3.14 is the cumulative representation of information shown in Figure 3.12. CFQ in the quest of being fair divides the I/O bandwidth (time slots) in round robin fashion amongst all the contending processes. This results in increase of the total number of disk head movements for serving the same I/O access requests, as different applications (processes) access data from multiple regions of the disk in a cyclic manner.

High rate of disk arm movements adversely affects the Service Level Agreements (SLAs) as well as the Total Cost of Ownership (TCO). It is extremely imperative to reduce the disk arm movements in-order to avoid disk failures (also the risk of data loss).

3.6.3.3 Disk Head Movement (seek) Distance

Figure 3.15 shows the average seek distance per disk head movement (AD_{seek}) in terms of number of disk cylinders (tracks) crossed. An interesting observation is that, CFQ outperforms all other schemes in most of the workloads. The main reason for the low average AD_{seek} is higher number of cumulative head movements "n" (see Figure 3.14), which occur due to round-robin nature of CFQ.

$$AD_{seek} = \frac{Cumulative\ Track\ Movement\ Distance\ \text{“}TMD_{seek}\text{”}}{Cumulative\ Head\ Movements\text{“}n\text{”}}$$

$$TMD_{seek} = \sum_{i=0}^{i=n} |SeekTrackNumber_{i+1} - SeekTrackNumber_i|$$

where, $SeekTrackNumber_i$ is the track or cylinder number of the i^{th} request. The error bars (standard deviation) for every scheduling scheme is fairly large. This is due to the nature of

distribution of disk arm movement distances. For the considered workloads, the disk arm movement distance is either much larger than mean distance or much smaller than mean distance. Therefore, the total distance traversed by the disk arm and number of disk head movements have no direct correlation. This is attributed to the skewness in the workload patterns as well as layout of the data stored in the disk, i.e. different applications store data in different zones (regions) in the disk.

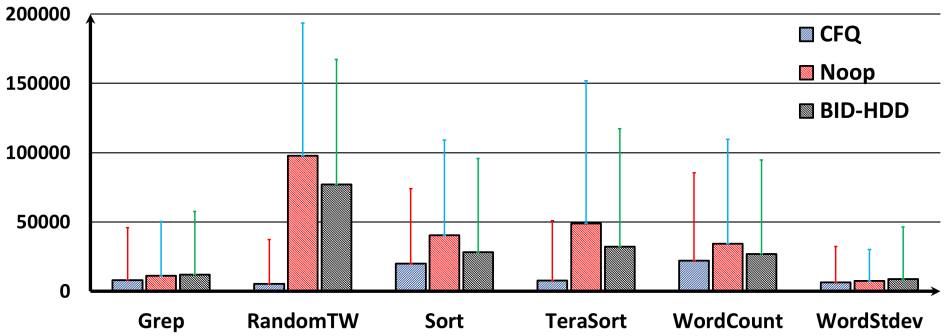


Figure 3.15: Avg distance (no. of cylinders or tracks) per disk arm movement.

Figure 3.16 shows the total seek distance (TD_{sweep}) traversed by the disk head in the course of serving all I/Os for different workloads. The distance is not shown in terms of number of tracks (cylinders), as the values tend to be very large. Instead, we chose the unit of distance to be one full disk sweep worth of tracks.

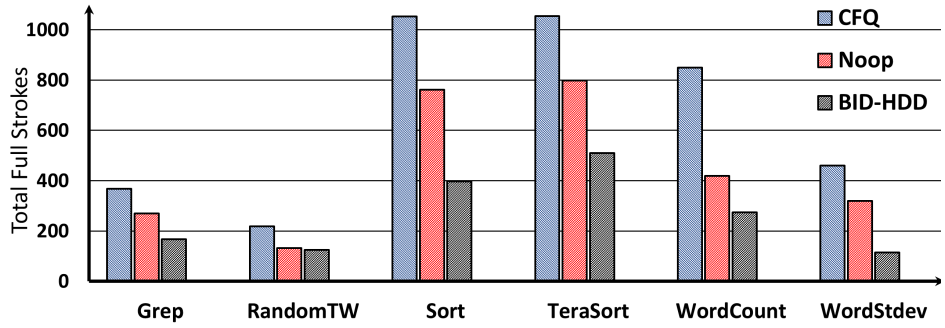


Figure 3.16: Cumulative disk head movement distance.

For example, consider the I/O submission order in Table 3.1. The output sequence (in terms of track numbers) by employing CFQ is as follows:

{3,3,16,71,(71,72),3,4,16,72,72,4,4}.

$$TMD_{seek} = |16-3| + |71-16| + |72-71| + |3-72| + |4-3| + |16-4| + |72-16| + |4-72| = 203$$

$$No.\ of\ head\ movements\ \text{“}n\text{”} = 8$$

$$AD_{seek} = \frac{TMD_{seek}}{n} = \frac{203}{8} = 25.37$$

$$Total\ No.\ of\ tracks\ (or\ cylinders)\ \text{“}Tr_{HDD}\text{”} = 100$$

$$TD_{sweep} = \frac{TMD_{seek}}{Tr_{HDD}} = \frac{203}{100} = 2.03$$

Thus, the total distance of disk arm movements to serve the *grep* workload when CFQ is employed is same as the distance the disk arm will move when HDD is sweeped fully for 380 times (refer to Figure 3.16). Figure 3.16 shows the overall impact of employing a scheduler. It shows that BID-HDD drastically reduce the total distance traversed by the disk arm. Similar justification of the amortization effect in BID-HDD, as discussed in previous sections, can be given for the reduction in disk arm movement distances. Distance traveled is related to the work done, or energy expended, therefore, BID schemes can also result in reduction in the energy footprint of storage systems.

It can be argued that the scheduler performance pattern observed in Cumulative I/O Completion Time, Figure 3.11 & Total distance covered Figure 3.16 should be similar. However, the relationship between distance traveled by disk arm and time taken is non-linear. For example, disk head movement between tracks 100 cylinders apart doesn't take 100 times the time taken between adjoining cylinders. There are few disk seeks which are non-preventable, which depend on which zone/region the contending applications store the data on the disk. The effect can be minimized by pipelining the requests as done by BID.

In future, we would like to combine BID with disk optimizations schemes like Borg [Bhadkamkar et al. (2009)]. The sparse locality of data belonging to different applications can be optimized by employing self-optimizing block reorganizing solutions like Borg [Bhadkamkar et al. (2009)]. Borg reorganizes blocks in the block layer based on workload I/O and LBA connectivities using graph

theory. Relocation of blocks in the disk drive take place on the principle of serving maximum I/O from dedicated partitions. Therefore, Borg could re-organize the blocks before submission to the I/O scheduler, while BID would optimize the contentions amongst the processes.

3.6.3.4 Impact on Individual Read/Write I/Os

In disks, higher throughput or lesser overall I/O time does not directly imply better I/O response times. It is important that I/O response times are lower as blocking I/Os (reads) force CPU to wait for the data from disk and suspend the process till it gets the data [Nanavati et al. (2015); Bhadkamkar et al. (2009); Joo et al. (2017); Bjørling et al. (2013)]. Moreover, reducing read (blocking I/O) latency is considered more important than reducing write (non-blocking I/O) latency. We discuss the impact of BID scheduling on read and write latency.

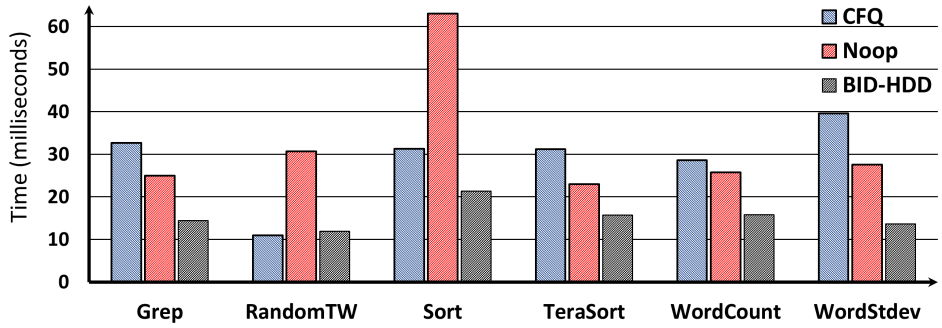


Figure 3.17: Mean Read I/O time.

Figures 3.17 and 3.18 show that on an "*average*", BID results in faster read and write I/Os performances for most (10 out of 12 cases) of the workloads. Noop also performs faster I/O for a majority (8 out of 12 of cases) of workloads. This is a very interesting result. On deeper analysis, we observe that due to the "*amortization effect*", BID-HDD might increase the staging or waiting time of few initial requests. The time taken to dynamically understand the I/O behavior of a process, make the initial I/Os of the sequential group experience a higher latency, however, due to lower latencies experienced by the later I/Os, their overall average latency is reduced.

This argument also explains why for *RandomTextWriter* and *WordStdev* BID-HDD results 10% slower read I/O than CFQ and 500% slower write I/O than CFQ, respectively. The reason for such

a behavior is that *RandomTextWriter* is highly sequential "write" workload with negligible number of small reads thus, the amortization effect does not come into play for read I/O. Moreover, as the number and sizes of reads are small, CFQ is able to serve them in a single time slice, while BID tend to wait for more requests and hence delay the reads. Same argument is valid for *WordStdev* in the case of write I/Os. BID-HDD would ensure the requests from each process would be served in the order in which the staging queues have expired. This could lead to higher serving time for small processes while in the case of CFQ or Noop, the wait might be smaller for such processes. Therefore, for processes which submit non-bulky I/Os that can be processed in a single time-slice, CFQ might be faster than BID-HDD for that process but again the over-all completion of all the processes (bulky and non-bulky) would suffer in the case of CFQ. Noop would be favored in cases when there are few processes (preferably only one active at a time), and each process submits large I/Os. BID-HDD would initially delay in staging and understanding the I/O characteristics, while Noop would directly send them to the device for processing. In a Big Data environment, this is a highly unlikely scenario due to multiple processes sharing the same resource, Noop does not scale well in such an environment.

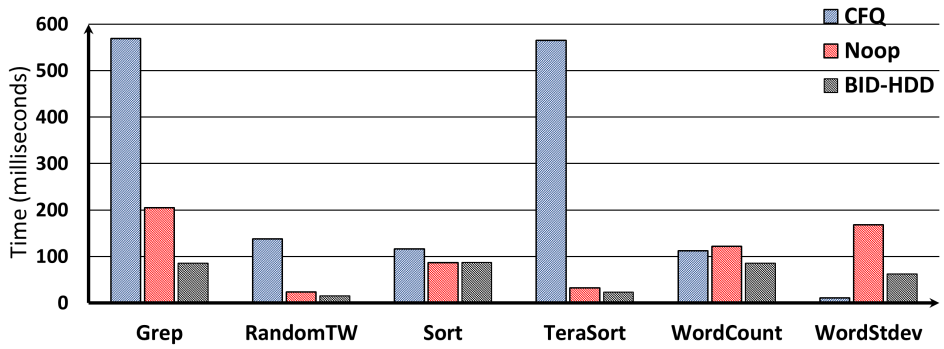


Figure 3.18: Mean Write I/O time.

BID is aimed to avoid contention following system constraints without compromising SLAs, as described in Section 3.3. Through trace driven simulation and experiments, we shows the effectiveness of both the schemes of BID, i.e. *BID-HDD* in shared multi-tenant, multi-tasking Big data cloud deployments. However in our experiments, we have shown the impact of block level

contention avoidance solutions on single physical storage device (HDD). The effect is additive when applied across all storage devices across the data center. BID essentially increases the efficiency of the block layer by streamlining the serving sequence of I/O requests to the block device in skewed, bulky and multiplexing I/O workloads like MapReduce. Other than resulting in faster I/O completion time, BID schemes can also result in increasing the lifespan expectancy of HDDs, data loss risk mitigation and energy savings due to reduction in the entropy of disk arm.

3.7 Conclusion

We have developed and designed a contention avoidance scheme for disk based storage devices known as "BID-HDD: Bulk I/O Dispatch" in the Linux block layer, specifically to suit multi-tenant, multi-tasking and skewed shared Big Data deployments. Through trace-driven experiments using in-house developed system simulators and cloud emulating Big Data benchmarks, we show the effectiveness of both our schemes. *BID-HDD*, which is essentially a block I/O scheduling scheme for disk based storage, results in 28% to 52% lesser time for all I/O requests than the best performing Linux disk schedulers. In future, it would be interesting to design a system with BID schemes for block level contention management coupled with self-optimizing block re-organization of BORG Bhadkamkar et al. (2009), adaptive data migration policies of ADLAM [Zhang et al. (2010)], and replication-management of such as Triple-H [Islam et al. (2015)]. This could solve the issue of workload & cost-aware tiering for large scale data-centers experiencing Big Data workloads.

Broader impact of this research would aid Data Centers in achieving their Service Level Agreements (SLAs) as well keeping the Total-Cost of Ownership (TCO) low. Apart from performance improvements of storage systems, the over-all deployment of BID schemes in data centers would also lead to energy footprint reduction and increase in lifespan expectancy of disk based storage devices.

CHAPTER 4. CONTENTION AVOIDANCE USING MULTIPLE TIERS

In this chapter, we discuss our novel hybrid scheme *BID-Hybrid* to exploit SCM's (SSDs) superior random performance to further avoid contentions at disk based storage compared to our contention avoidance solution, *BID-HDD* discussed in Chapter 3. *BID-Hybrid* [Mishra and Somani (2017)] is able to efficiently offload non-bulky interruptions from HDD request queue to SSD queue using BID-HDD for disk request processing and multi-q FIFO architecture for SSD. This results in performance gain of 6% to 23% for MapReduce workloads when compared to BID-HDD and 33% to 54% over best performing Linux scheduling scheme. BID schemes as a whole is aimed to avoid contentions for disk based storage I/Os following system constraints without compromising SLAs.

The chapter is organized as follows. First, we discuss the associated issues of disk based storage devices (HDDs) and its impact on performance in Section 4.1. Section 4.2 provides a brief overview of SCM (Storage Class Memory) devices characteristics, its advantages as well as factors which inhibit their complete adoption in Data Centers. Section 4.3 discusses in brief the additional features of the block layer and need for a hybrid-aware block layer. In Sections 4.4 and 4.5, we present the architecture of our Contention Management scheme i.e. **BID-Hybrid** with its various components followed by our design of experiments and performance evaluation, respectively. Section 2.1.2 discussed in Chapter 2 provides an in-depth survey of related literature. We conclude the chapter in Section 4.6 with a discussion on future work.

4.1 The Problem

HDDs form the backbone of Data Center storage, but due to their physical limitations and I/O characteristics of the workloads (multiplexing of concurrent applications I/Os, phase I/O profile, etc.), the performance of HDDs suffers due to physical movement of disk arm to access data. The inherent sequentiality is lost due to I/Os from some processes (or phase of process), which perform

non-bulky or non-sequential I/O accesses. Despite of having *"ideal"* (Refer to Section 3.4), or the most sequential I/O scheduler employed, the sequential performance gains of HDDs cannot be derived. In-order to achieve maximal sequentiality for HDDs, it is imperative to factor out the random I/Os from the HDD request queue. The main question to answer is *how* to profile such processes/ applications on-the fly during data placement due to the time-varying I/O characteristics of such processes. In this chapter, we design and develop a system comprising of HDDs and SSDs, which deterministically determines such interruption causing I/Os and utilizes heterogeneous *tiers*[1] of storage to make data placement decisions.

With recent developments in NVMe (Non-Volatile Memory) devices such as Solid State Drives (SSDs), commonly known as Storage Class Memories (SCM) [Mittal and Vetter (2016)], with supporting infrastructure, and, virtualization techniques, a hybrid approach of using heterogeneous tiers of storage together such as those having HDDs and SSDs coupled with *workload-aware tiering*[2] to balance cost, performance and capacity have become increasingly popular [**?**Krish et al. (2016)]. We propose a novel hybrid scheme *BID-Hybrid* to exploit SCM's (SSDs) superior random performance to further avoid contentions at disk based storage. The main goal of BID-Hybrid is to further enhance the performance of BID-HDD scheduling scheme, by offloading interruption causing non-bulky I/Os to SSD and thereby making the "HDD request queue" available for bulky and sequential I/Os.

Contrary to the existing literature of tiering, where data is tiered based on deviation of adjacent disk block locations in the device "request queue", BID-Hybrid profiles process I/O characteristics (bulkiness) to decide on the correct candidates for tiering. The current literature might cause unnecessary deportations to SSDs, due to I/Os from an application, which might be sequential but appear random due to the contention by other applications in submitting I/O to the "request queue". While BID-Hybrid uses staging capabilities and anticipation time for judicious and verified decisions. BID-Hybrid serves I/Os from bulky processes in HDD and tiers I/Os from non-

[1]Storage media across all nodes with similar I/O performance or characteristics form a *tier* (or tiers of storage).
[2]While *tiers* or *Tiering*, when used as an adjective, refers to orchestrating data between heterogeneous tiers of storage by leveraging individual strengths of each to maintain balance between Cost, Performance and Capacity.

bulky (lighter) interruption causing processes to SSD. BID-Hybrid is successfully able to achieve its objective of further reducing contention at disk based storage device. BID-Hybrid results in performance gain of 6% to 23% for MapReduce workloads over BID-HDD and 33% to 54% over the best performing Linux scheduling schemes. BID schemes (BID-HDD and BID-Hybrid), as a whole are designed to avoid contentions for disk based storage I/Os following system constraints without compromising SLAs. In the next section, we discuss the details of devices with superior random I/O performance and their applicability in the data storage hierarchy.

4.2 Storage Class Memory (SCM) characteristics

Due to the physical limitations of HDDs, there have been recent efforts [Zheng et al. (2013); Krish et al. (2016); Kim et al. (2016); Moon et al. (2015); Iliadis et al. (2015); Islam et al. (2015); Krish et al. (2014b)] in incorporating flash based storage such as SSDs in data centers. The high-speed, non-volatile storage devices like SSDs typically referred to as Storage Class Memories (SCMs) access data via electrical signals, as opposed to physical disk arm movement in the case of HDDs [Mittal and Vetter (2016); Nanavati et al. (2015)]. Data is organized in 4 KB pages, while a group of 128 or 256 pages is known as block in SCMs. The data in SCMs is written at granularity of pages but the deletion happens at the block granularity.

Despite superior random performance of SCMs (or SSDs) over HDDs, replacing slower disks with SCMs doesn't seem to be economically feasible for data center applications [Mittal and Vetter (2016); Krish et al. (2016)]. Few of the major disadvantages of SCMs are enumerated below:

- *Cost and Lifespan:* The main disadvantage of SCMs over HDD is their high cost and limited lifespan. SCMs can endure limited write-erase cycles, i.e. after a threshold of writes, the pages becomes dysfunctional. Therefore SCMs increase the Total Cost of Ownership (TCO). Unless majority of the data is uniformly "hot" it is highly inefficient to store the data in high-value SCMs as they are underutilized and do not justify the high investment [Zhou et al. (2016); Harter et al. (2014); Nanavati et al. (2015)].

- *Write Amplification:* To increase life-time of SCM pages, the firmware tries to spread the writes throughout the device. Additionally, due to physical constraints, SCMs cannot over-write at the same location (page). As deletion or erase happens in the granularity of blocks, therefore a single page update requires a complete block erase and out-of-place write. These result into unwanted phenomenons such as write amplification (wear-leveling) and garbage collection (faulty block management) [Yang and Zhu (2016b); Mittal and Vetter (2016); Moon et al. (2015)]. These activities consume a lot of CPU time as well as the SSD controller and the File System have additional jobs such as book-keeping than simple data access.

- *Skewed Write Performance:* The superior performance of SCMs over HDDs is highly dependent on the workload. For write-intensive scientific and industrial workloads, the performance of HDDs and SSDs have been shown to be nearly same [Mittal and Vetter (2016)]. The skew in performance makes it more economically feasible to use HDDs.

SCMs used to work on legacy disk based interface such as SATA/SAS. Recently there has been great industrial and academic focus to utilize faster PCIe bus technology (also known as NVMe Express) as an interface for SSDs [Nanavati et al. (2015)]. NVM Express is becoming the de-fact standard to interact with SCMs over PCI Express. NVMe over RDMA (fabrics), PCIe switches, Linux block layer redesign are few of the solutions being developed for enabling NVMe express driver for data-centers [Malladi et al. (2016); Eshghi and Micheloni (2013)].

There are additional drawbacks such as lack of aligned software stack to utilize the internal parallelism of SCMs as well problems associated with interface and channel sharing. There has been a paradigm shift in modern data-center to adopt a hybrid approach of using heterogeneous storage devices such as HDDs and SCMs. Therefore in most existing literature, SCMs are used as cache for disk based storage, coupled with workload-aware tiering [Nanavati et al. (2015); Zhou et al. (2016); Krish et al. (2016); Roussos (2007)] for automatic classification of data to balance cost, performance and capacity. While extending the concepts of memory-cache hierarchy to the HDD-SCM pair seems to be logical step for such large working sets, but this has its pros and cons, mainly attributed to basic design of such systems, i.e. the caching algorithms. Similar to

the problems associated to caching, the performance gains are based on prediction based heuristics depending on popularity, frequency or deviation of addresses, which may or may not be beneficial. Some data blocks might be popular in one phase while they might not be after being migrated or placed to a higher tier. Moreover, the time-varying nature of applications make it extremely difficult to profile the correct candidates for tiering. All such caching-based mechanisms are based on prediction, which may be ineffective in many cases. While our approach to tiering, **BID-Hybrid** uses the knowledge of the OS kernel I/O data-structures to aid deterministic tiering decisions for Big Data deployments. In the next section, we discuss the additional features of the OS block layer to support such as system and the need for development of hybrid-aware block layer for tiering.

4.3 OS Block Layer: Additional Features and Need for Hybrid-Awareness

Software defined storage (SDS) is the means of delivering storage services for a plethora of data center applications and environments. Storage virtualization is the building block for SDS as it aids in provisioning storage (LUN, LVMs, etc) with heterogeneous devices, automated tiering, increasing storage utilization and providing software solutions for data management.

In order to deliver SDS for current and future needs of Big Data, apart from efficient tuning up of the block layer's current capabilities like the I/O scheduler, I/O data-structure management, accounting etc., additional functionalities like automated workload based tiering, etc. need to be added. The importance of the block layer in the I/O stack for its role in managing I/Os and resolution of contentions amongst applications (*I/O Scheduling*) is discussed in detail in 3. Additionally, the block layer has the following benefits which make it suitable for developing SDS solutions, especially automated tiering:

- *Hardware Agnostic:* The block layer is the point of entry for I/O requests for a block storage device (HDDs or SSDs), except for direct I/Os (Direct Memory Access) which are handled by strict interrupts. Any solution or optimization in the block layer can be applied to a diverse range of storage devices, making the solution independent of the underlying physical media.

- *File System Agnostic:* The block layer lies below the VFS and above the device driver. Operating at the block layer makes the solution independent of the file system layer above, enabling it with the flexibility to support multiple heterogeneous file systems simultaneously [Bhadkamkar et al. (2009); Bjørling et al. (2013)].

- *Information Capturing:* Accounting information such as block, file and process the requests belong to, is isolated from the device driver, as the function of the device driver is only to transmit the requests from the block layer to the physical storage. The Block Layer has access to such attributes [Bhadkamkar et al. (2009); Bjørling et al. (2013)], thereby providing opportunities to exploit information for intelligent optimizations. Additionally, the kernel I/O sub-structure constructions (refer Appendix A) and stage transformation information is available in the block layer. These are critical as above and below the block layer, such information are hidden [Bhadkamkar et al. (2009)].

- *Storage Architecture Agnostic:* Irrespective of storage networked virtualizations like SAN, NAS or DAS, ultimately I/Os are managed by the block layer. The block layer as shown in Figure 3.1 resides towards the client in centralized storage management solutions like SAN, while in the case of NAS, it lies inside the NAS device. Therefore, any solution built in the block layer can be applied to any data-center storage infrastructure.

Therefore, the universal applicability and the inherent I/O information possessed by the OS block layer coupled with system architecture could be harnessed to make deterministic tiering decisions.

As discussed earlier, despite the significance of the OS block layer, there hasn't been enough development to adapt to the ever changing paradigm of big data. The block layer for disk based storage (HDDs) has still remained highly volatile as the mechanical disks cannot support multiple hardware queues due to their physical constraints. Therefore, HDDs can have multiple software queues but single Hardware queue. Very recently, an important piece of work [Bjørling et al. (2013)] tries to extend the capabilities of the block layer for utilizing internal parallelism of NVMe SSDs to

enable fast computation for multi-core systems. It proposes changes to the existing OS block layer with support for per-CPU software and hardware queues for a single storage device. It is imperative to develop solutions to harness the potential of such multi-queue block layer architecture.

The objective function of block layer for disk based storage is to optimize the request order from various applications in-order to recreate sequentiality of disk access and manage the I/O bandwidth for every application. In **BID-HDD** (refer Section 3.5), we design and develop a block layer with multiple software queues (per process) and single hardware queue (device driver) for disk based storage. Our block I/O scheduling scheme in BID-HDD, Algorithm 1 uses the multiple software queues to profile I/O contending processes and flushing I/O requests in sequence to the single dispatch hardware queue (using Algorithm 2). Thereby, recreating sequentiality of I/O accesses and improving performance.

Despite the best effort or "ideal" block I/O scheduling scheme, there are interruptions (which occur and reoccur), which are not sequential or belong to a phase of a process which accesses data in small data blocks. Such I/O accesses cause disk arm movements, which are small in I/O size but impact the performance largely, when placed in the HDD request queue. Therefore, its essential to identify such performance critical blocks and place them in a physical media such as SSDs which have superior random I/O performance. These further avoid contentions in the HDD request queue and improve performance during current access as well as future references to such blocks. We develop and design, **BID-Hybrid**, to work in a multi software queue architecture similar to BID-HDD. It uses BID-HDD for staging capabilities to dynamically profile the I/O of each process to determine the correct candidates for tiering to make decisions for initial data placement. Once such decisions are made, the HDD queues follow BID-HDD block I/O scheduling, while the SSD queues follow multi-q [Bjørling et al. (2013)]. This makes the block layer hybrid aware, which is essential to identify and place the performance critical I/Os.

Therefore, BID schemes utilize multiple software queues in the block layer, but single hardware queue for delivering SDS solutions for disk based storage devices. In the next section, we discuss Our Approach to tiering, i.e. **BID-Hybrid**.

4.4 Our Approach to *tiering*: BID-Hybrid

We propose a hybrid scheme *BID-Hybrid* to exploit SCM's (SSDs) superior random performance to further avoid contentions at disk based storage. In the hybrid approach, dynamic process level profiling in the block layer is performed for deciding the candidates for tiering to SSD. Therefore, I/O blocks belonging to interruption causing processes are offloaded to SSD, while bulky I/Os are served by HDD. BID-HDD scheduling scheme is used for disk request processing and multi-q [Bjørling et al. (2013)] FIFO architecture for SSD I/O request processing.

BID schemes are designed taking into consideration the requirements laid out earlier in Section 3.3. BID as a whole is aimed to avoid contentions for storage I/Os following system constraints without compromising the SLAs.

Due to physical limitation of HDDs, there have been recent efforts to incorporate flash based high-speed, non-volatile secondary memory devices, known as Storage Class Memories (SCMs) in data centers. Despite superior random performance of SCMs (or SSDs) over HDDs, replacing disks with SCMs completely for data center deployments doesn't seem to be feasible economically as well as due to other associated issues discussed briefly in Section 4.2 [Mittal and Vetter (2016); Krish et al. (2016)].

With recent developments in NVMe (Non-Volatile Memory) devices, with supporting infrastructure, and, virtualization techniques, a hybrid approach of using heterogeneous tiers of storage together such as those having HDDs and SSDs (Solid State Drive) coupled with workload-aware tiering to balance cost, performance and capacity have become increasingly popular]Krish et al. (2016); Nanavati et al. (2015); Malladi et al. (2016); Roussos (2007)].

Data centers consists of many tiers of storage devices. All storage devices of the same type form a *tier* [Krish et al. (2014b)]. For example: all HDDs across the data-center form the *HDD tier* and all SSD form *SSD tier*, and similarly for other SCMs. Based on profiling of workloads, balanced utility value of data usage, the data is managed between the tiers of storage for improved performance. Workload-aware Storage Tiering, or simply *Tiering* [Zhou et al. (2016); Nanavati et al. (2015)] is the automatic classification of how data is managed between heterogeneous tiers of

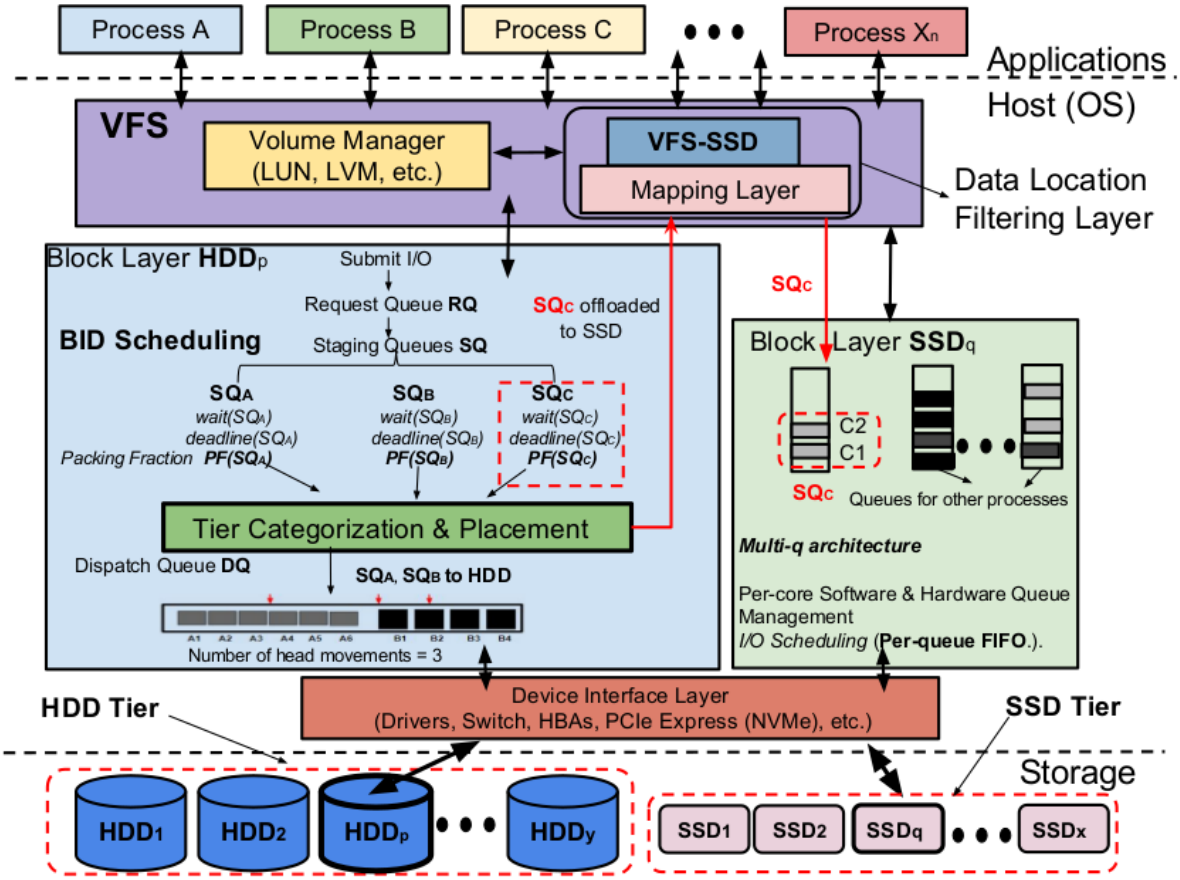
storage in enterprise data-center environment [Zhang et al. (2010)]. It is vital to develop automated and dynamic tiering solutions to utilize all the tiers of storage. BID-Hybrid aims to deliver the capability of dynamic and judicious automated tiering in the block layer as a Software Defined Storage solution.

BID-Hybrid is designed to suit multi-tasking, multi-user shared Big Data environments. Contrary to the tiering approach of defining SSD candidates based on deviation of LBAs, BID-Hybrid profiles process I/O characteristics by utilizing dynamic anticipation and I/O packing. BID-Hybrid uses similar concepts of staging as BID-HDD. Due to the staging capabilities in the host (OS) block layer, bulkiness of processes can be calculated and verified on-the fly in-order to avoid unnecessary deportations to SSD. The key idea is to offload I/O blocks belonging to non-bulky processes to SSD (managed by multi-q block layer architecture in Bjørling et al. (2013)) and the bulky I/Os to HDD (handled by BID-HDD discussed in Chapter 3). This serves multi-fold: 1) maximal sequentiality in HDD is ensured, i.e "HDD request queue" is made free from unnecessary contention and interruption causing blocks; 2) the future references to the non-bulky blocks are prevented from causing contentions for HDD disk I/O, as the semantic blocks have a high probability to appear in the same pattern [Bhadkamkar et al. (2009); Ibrahim et al. (2011); Chen et al. (2011)]. Therefore, BID-Hybrid aims to further reduce contention (more than BID-HDD) at disk based storage by offloading interruption causing blocks to SSD, while ensuring uninterrupted sequential access to HDDs.

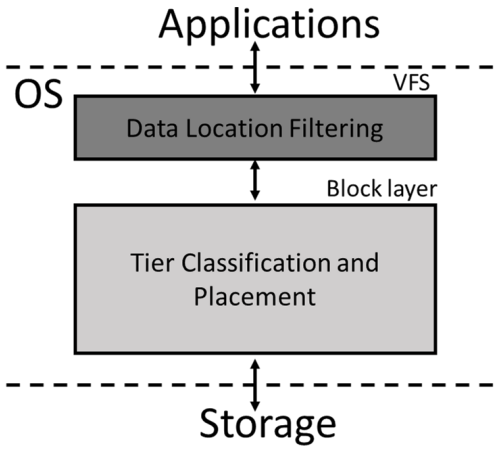
4.4.1 Architecture

In Figure 4.1a, we show the system architecture of BID-Hybrid in multi-tier storage environments with the help of the I/O submission order as in Table 3.1. To show the effectiveness in offloading non-bulky requests to SSD during initial write as shown in Table 4.1.

The VFS, Volume Manager, Mapping Layer and VFS-SSD is shown in Figure 4.1a as a single layer, to imply that it spans across the cluster and provides the storage virtualization functionalities of abstracting data locality from the applications and unwrapping data location for execution of



a) System architecture of *BID-Hybrid*.



b) Components of *BID-Hybrid*.

Figure 4.1: Working architecture of *BID-Hybrid*.

Table 4.1: I/O request Submission Order to the Hybrid-aware Block Layer.

order	request	LBA	transfer size	Track No. (cylinder)	read/write	time to expire (ms)
1	B1	7125	40	71	w	Exp#1
2	A1	305	24	3	r	Exp#2
3	A2	340	24	3	r	Exp#3
4	A3	370	24	3	r	Exp#4
5	C1	1600	4	16	w	Exp#5
6	B2	7165	40	71, 72	w	50
7	B3	7205	40	72	w	53
8	A4	410	24	4	r	60
9	A5	440	24	4	r	65
10	A6	470	24	4	r	100
11	C2	1670	4	16	w	105
12	B4	7245	40	72	w	110

I/O to the appropriate storage device. A file can span across multiple storage devices but appear to the applications to be stored on a single device.

BID-Hybrid modifies the block layer (extend the capabilities of BID-HDD modifications) in-order to take tier-placement decision as well as leverage storage virtualizations such as Virtual File System for infrastructural support for offloading and locating tiered SSD blocks. The working architecture of BID-Hybrid consists of two major modules (consider Figure 4.1b), i.e. 1) Data Location Filtering and 2) Tier Classification and Placement, which are discussed in Sections 4.4.2 and 4.4.3.

4.4.2 Data Location Filtering

Even before tier classification decision is made, there is a need to filter the requests which are already written. The reason being that the classification of previously written data has already been done during previous accesses. BID adds a *Data Location filtering layer* to the VFS, which consists of a sub-module, **VFS-SSD** that records the location information (SSD block device, page number, LBA re-addressing etc.,) in a table to keep track of the previously tiered (written) data. The VFS-SSD works with the Logical Volume Manager and the Mapping Layer to filter and find the tiered data from the requests submitted to the VFS by the applications. Any future reference (read or update) to the already tiered data is handled by VFS-SSD. If the I/O request is found in

the VFS-SSD table, it is enqueued to the block layer of the respective SSD device. After the data location filtration, those I/O access requests which are not present in VFS-SSD table are enqueued to the block layer of the respective HDD.

4.4.3 Tier Classification and Placement

Those accesses intended for HDDs, follow the steps exactly as BID-HDD, i.e. enqueuing I/O request in *request queue* RQ, dequeuing I/O request from RQ to respective process *staging queue* SQ_P, compute the wait and deadline timer, mark staging queue SQ_P for flushing (see Section 3.5 & **Algorithm 1**). Once the staging queue SQ_P is marked for flushing, the tier placement & categorization decisions for the writes is performed. The placement decisions are made once the anticipation time $wait(SQ_P)$ *or* $deadline(SQ_P)$ *timer* has expired. Each *staging queue* SQ_P has ample time to merge adjoining requests as well as processes also have time to submit I/Os to the request queue RQ. This makes the profiling of *staging queue* SQ_P's more judicious, thereby making tier categorization decisions more accurate. Please note, the read only staging queues by-pass the Tier-Categorization and Placement layer as they are those requests which have already been written and the tier placement decision had considered them HDD favorable.

For performing the tiering classification, BID-Hybrid uses a quantity called *Packing Fraction* $PF(SQ_P)$ for determining the bulkiness of any process. $PF(SQ_p)$ is associated with every staging queue SQ_P and is defined as the ratio of cumulative I/O access size of all write requests (in units of 512 byte disk blocks) and the total number of write requests present in SQ_P (in terms of "request" kernel I/O structures). The tier placement & categorization decision and dispatch of I/Os follows Algorithm 3.

$$Packing\ Fraction\ PF(SQ_P) = \frac{Cumulative\ I/O\ request\ size_{write}(SQ_P)}{Total\ No.\ of\ requests_{write}(SQ_P)\ \text{“}k\text{”}}$$

$$Cumulative\ I/O\ RequestSize_{write}(SQ_P) = \sum_{i=1}^{i=k} size\ of\ write\ request_i(SQ_P)$$

ALGORITHM 3: Tier Categorization and Placement Decision

for *every staging queue* SQ_p *marked for flushing*
select the one, say SQ_P *which was marked earliest.* **do**
  if $PF(SQ_p) < PF(Threshold)$ **then**
    I/O requests in SQ_p are SSD favorable;
    Transfer write I/O requests to SSD I/O request queue;
  Mark SQ_p for flushing to HDD dispatch queue;

Key intuition: It is based on the working of the block layer and kernel sub-structures in coalescing maximum adjoining BIO (block I/O) structures in a request structure. Big Data applications access data in large chunks. For example, MapReduce processes access data in 64 MB HDFS chunks. Therefore, the resultant "request" structures tend to be bulky (more data per request), i.e. high *Packing Fraction* $PF(SQ_P)$. However due to time varying I/O characteristics and nature of application some MapReduce applications might have stages (processes) in which data accesses are small and random (for eg: small intermediate writes subsequent reads, shuffle and combine intermediate data, etc.) [Ibrahim et al. (2011); Harter et al. (2014)]. Therefore, the resultant I/O "request" structures tend to be lighter or non-bulky, i.e. have low *Packing Fraction* $PF(SQ_P)$. The I/Os from non-bulky light processes, culminate into increasing contentions resulting into breaking the sequentiality of I/Os from bulky processes. Moreover, future references to these interruptions have a high probability to occur in the same fashion [Harter et al. (2014)].

Once the tiering decisions are made, the bulky or HDD favorable staging queues are marked for flushing to the HDD dispatch queue. The flushing of the HDD favorable staging queues follows the pipeline as described in **Algorithm 2**. The management of non-bulky SSD favorable I/O requests is done as follows:

The transfer of data is managed as per the tier migration model, i.e. the Host (OS) initiates a process to transfer the I/O requests belonging to non-bulky or SSD favorable staging queues via the network through peer-to-peer data transfer protocol to the Host (OS)-of targeted SSD. Storage virtualization provides additional features for movement of data between tiers and machines via efficient inter-connect technologies such as RDMA (Remote Direct Memory Access), Infiniband, RCoE (RDMA over converged Ethernet), etc. [Pfefferle et al. (2015)].

We have considered the case where dedicated SSDs (Solid State Drives) are used for storing the non-bulky data accesses. The *VFS-SSD module* is also responsible to map dedicated shared SSDs and provision available SSDs for tiering according to the topology. Multiple HDDs can share a single SSD as the non-bulky data per HDD is usually small. Each non-bulky staging queue is spawned on the SSD block layer as a separate process submitting I/O, so BID-Hybrid uses the Multi-q architecture as described in Bjørling et al. (2013) employing a FIFO per queue scheduling scheme.

Consider Figure 4.1a, amongst the I/O access requests for processes *A, B, C*, *staging queue* SQ_C is found to be an ideal candidate for tiering. The I/O requests for process C are determined to be "non-bulky", due to its low *Packing Fraction* PF_C. While processes *A and B* are determined to be bulky. Therefore, the *staging queues* SQ_A & SQ_B flush request as per BID-HDD, while requests belonging to process *C* is managed by the SSD block layer using Multi-q [Bjørling et al. (2013)] architecture via per queue FIFO based scheduling.

Using the above for contention avoidance storage solutions, BID schemes are capable of delivering higher performance. In the next section, through trace-driven simulation experiments using cloud emulating Hadoop benchmarks, the performance of BID-HDD and BID-Hybrid is evaluated and compared with the current Linux scheduling schemes.

4.5 Experiments and Performance Evaluation

Similar to BID-HDD in Chapter 3, we demonstrate the effectiveness of BID-Hybrid through trace-driven simulation and additions to our in-house developed system simulators (refer to Section 3.6.2) to conduct experiments for performance evaluation.

The testbed setup, trace-collection and cloud emulating Hadoop data-centric workloads are same as discussed in Section 3.6.1. In the following section, we describe only the addition to the system simulator (refer to Section 3.6.2) required to evaluate BID-Hybrid. In the end, we discuss the performance evaluation of BID-Hybrid in Section 4.5.2 following the description of our in-house developed hybrid system simulator.

4.5.1 Hybrid System Simulator

We have added modules and made relevant changes to the system simulator as discribed in Chapter 3 using Python v2.7.3 to replicate the working of a hybrid storage aware system. We use the trace file (as discussed in Section 3.6.1) for application I/O submission order. The Simulator has three major modules: a) *VFS Simulator:* Performs Virtual File System for locating and book-keeping; b)*OS Simulator:* Takes the order of I/O submissions and performs Linux Kernel block layer functions (contains pluggable I/O Scheduler sub-module); and c) *Storage Simulator:* Takes input from OS Simulator and returns performance metrics. The details of each of these components (see Figure 4.2) and the modifications to Section 3.6.2 is discussed below.

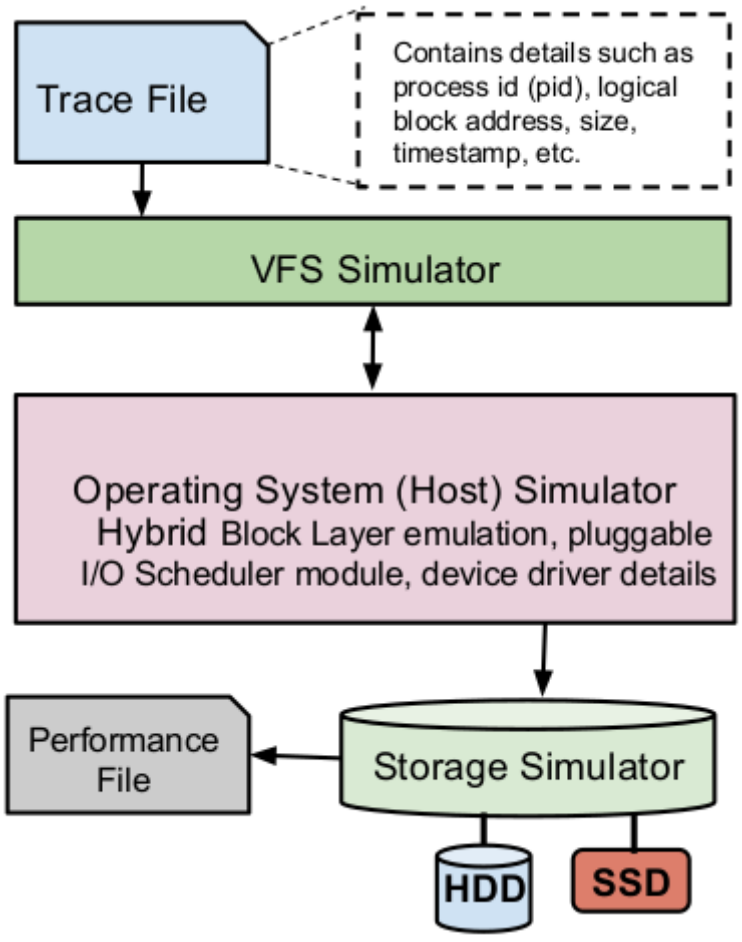


Figure 4.2: Simulator Components for Hybrid Aware Storage Systems.

- **OS Simulator:** This module takes the collected workload I/O traces (Trace File) as input and recreates the Kernel Block Layer functions after the stage from which the traces were collected (refer to Section 3.6.1). The OS Simulator performs the same functionalities as the OS simulator in Section 3.6.2 with added features for Hybrid aware block layer. In the case of the Hybrid Approach, the OS Simulator 1) makes Tier Placement decisions; 2) interacts with the VFS to map the LBA entries offloaded to SSD for future reference; 3) spawns the process on the Block Layer for appropriate SSD as per the multi-queue architecture [Bjørling et al. (2013)]. To preserve the I/O characteristics of the workloads, the requests are submitted based on the timestamp from the trace file to the kernel block layer.

- **VFS Simulator:** The main function of the Virtual File System (VFS) is to locate the blocks required by applications. The VFS and the Mapping layer along with abstractions such as logical volume manager provide storage virtualization. This enables storage pooling, capacity utilization and unifying storage with heterogeneous devices which aids data migration and placement policies. As the traces already have the LBA and targeted block device, so the simulator is designed to work in case of Hybrid "tier placement" decisions, as the change of target device is taken on the fly. The VFS-SSD sub-module stores in a table, which data is stored in which block device. This aids in finding the location of future reference to already written data.

- **Storage Simulator:** This module takes the I/O requests from the dispatch queue of the OS Block Simulator and based on the device type (HDD or SSD), return performance metrics like completion time depending on the current state of the block device. The module takes block device configuration parameters as inputs (device driver) such as drive capacity, block device type (HDD or SSD), etc. For SCMs (SSDs), the drive parameters include the number of pages per block, size of each page, seek time (read, writes, erase) etc. The Storage simulator calculates the I/O access time (per I/O request) by HDDs as discussed in Section 3.6.2. For SSDs, the access time depends on the SSD properties provided by the manufacturer. The

configurable features gives us the ability to test the schemes with different devices as well as drive architectures.

Using the simulator we evaluate BID-Hybrid to show the improvements with respect to BID-HDD, and the current Linux block I/O schemes discussed below.

4.5.2 Performance Evaluation: Results and Discussions

We compare the effectiveness of our Hybrid Contention Avoidance solution BID-Hybrid with BID-HDD and the two best performing Linux block I/O Scheduling scheme. In Chapter 3, we have already shown that BID-HDD outperforms the two best performing Linux schemes, i.e. CFQ and Noop. Out motive here is to show the improvements BID-Hybrid brings to further reduce contentions at the disk interface over BID-HDD. For our experiments, we use the default parameters depicted in Table 4.2, which are based on the storage devices and driver specifications.

Table 4.2: Block Device Parameters for Hybrid storage systems evaluation.

Block Device	**Default Parameters**

HDD	maximum "request" structure size = 512 KB;
	request queue size = 256 BIO structures (128 reads, 128 writes);
	max. size of each block I/O (BIO) structure = 128 x 4K pages;
	1 page (bio vec) = 8 x 512-byte disk sectors (block);
	access granularity (disk block sector size) = 512 bytes.
	Specification based exactly as our 250 GB Hadoop cluster HDD.

SSD	block size = 256 pages; page size = 4KB = access granularity;
	buffer access time (random 4K): read = 0.025 ms; write = 0.5 ms;
	Specification based on SLC Flash SSD in [Mittal and Vetter (2016)].

The performance evaluation of the contention avoidance schemes is discussed in the following sections.

4.5.2.1 Cumulative I/O Completion Time

Figure 4.3 represents the cumulative time taken (x-axis) by the block device (in case of Hybrid approach, devices) to fulfill all the I/O requests using the different schemes. In Section 3.6.3.1, we have shown the effectiveness of BID-HDD over the Linux scheduling schemes, i.e. CFQ and Noop. Here, we discuss the further time savings and improvements in avoiding contentions for disk based storage using BID-Hybrid as compared to BID-HDD.

Figure 4.3 demonstrates that BID-Hybrid is able to further improve the performance of BID-HDD for all workloads by 6 to 23%, with maximum gain in the case of *WordCount* workload (see Table 3.2). In *WordCount*, the sequentiality of the reads is preserved due to the displacing of interruptions, i.e. large number of small writes, which would have contended for HDD I/O time. The bulky processes are handled in HDD via BID-HDD scheduling scheme, while the non-bulky (small) write I/O submitting processes are deported to SSD (by Tier Categorization Layer), which is served by the Multi-queue [Bjørling et al. (2013)] block layer of SSD. This serves multi-fold, first maximal sequentiality in HDD is ensured, i.e. preventing the avoidable disk arm movements (interruptions). Secondly, preventing future interruptions in HDDs sequentiality, as the blocks which are non-bulky have a high probability to appear in the same pattern [Bhadkamkar et al. (2009); Ibrahim et al. (2011); Chen et al. (2011)]. This in turn would lead to further smoothening of the BID-HDD graph of Figure 3.13, with spikes in the graph being leveled.

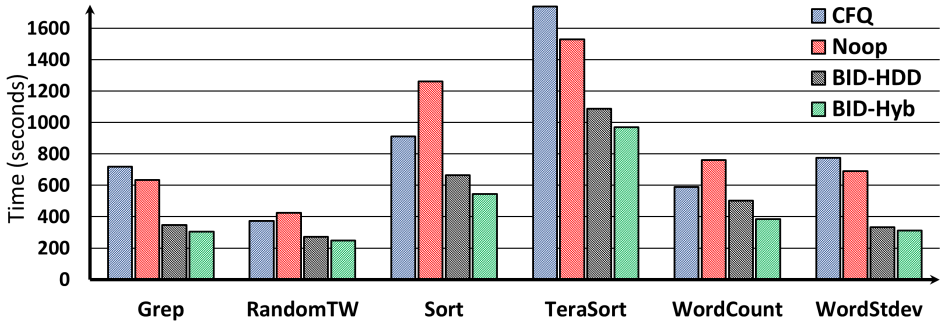


Figure 4.3: Cumulative I/O Completion Time.

Takeaway: BID-Hybrid is further able to reduce the contention at the HDD block layer by taking displacement decisions for small (in terms of I/O request size) but performance incongruous I/O requests to SSD. Thereby providing more opportunity to sequentialize processes submitting bulky I/Os as well as avoiding preventable disk seeks. The impact of this reduced I/O access time on total application execution time can be much higher, as the CPU wait times is reduced [Joo et al. (2017); Bhadkamkar et al. (2009); Bjørling et al. (2013)].

4.5.2.2 Number of Disk Arm Movements

Figure 4.4 shows the disk arm movements incurred by all workloads. Earlier in Section 3.6.3.2, we have discussed how BID-HDD outperforms the Linux schemes. BID-Hybrid leads to fewer disk head movements as compared to BID-HDD due to the offloading of non-bulky requests to SSD from the HDD request queue. The justifications is similar as discussed in Section 3.6.3.1 and Section 4.5.2.1, i.e. the amortization affect of BID-HDD attributes to the reduction in disk arm movements,

.

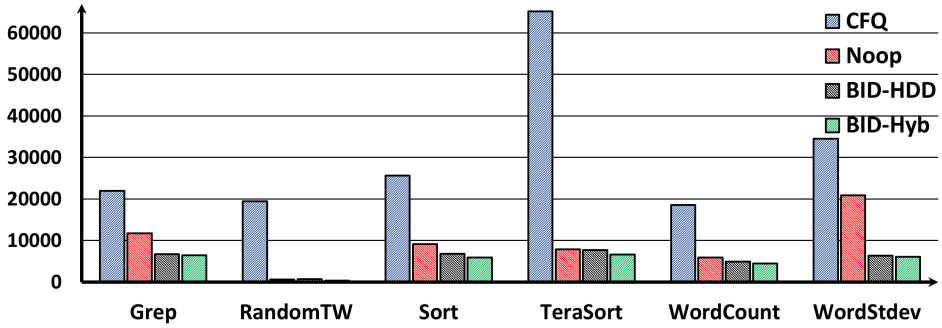


Figure 4.4: Total number of disk arm movements.

In BID-Hybrid, we observe that the disk head movement reduction w.r.t. BID-HDD, is maximum in workloads like *TeraSort* (*gain* 50%), with mixed I/O characteristics, Sequential reads/writes along with large number of small (random) reads/writes. Therefore, BID-Hybrid is able to successfully capture the deviation causing lighter I/O accesses during the initial tier placement (write) and offload them to SSD. These lighter I/O requests potentially could have adversely affected the sequentiality of bulky processes. They also prevents future contentions on HDD request queue

arising from these I/O accesses, in turn providing the higher opportunity to maintain the inherent sequentiality.

High rate of disk arm movements adversely affects the Service Level Agreements (SLAs) as well as the Total Cost of Ownership (TCO). It is extremely imperative to reduce the disk arm movements in-order to avoid disk failures (also the risk of data loss).

4.5.2.3 Disk Head Movement (seek) Distance

Figure 4.5 shows the average seek distance per disk head movement (AD_{seek}) in terms of number of disk cylinders (tracks) crossed. The justifications regarding the reduction in disk head movement for BID-Hybrid is similar to BID-HDD which has been discussed in detail in Section 3.6.3.3.

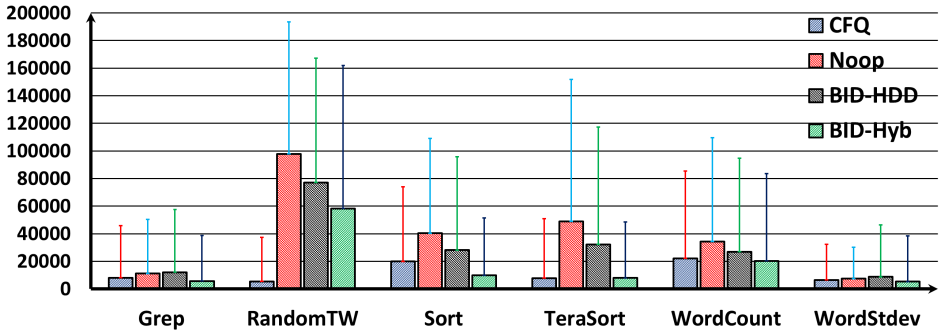


Figure 4.5: Avg distance (no. of cylinders or tracks) per disk arm movement.

Figure 4.6 shows the overall impact of employing a hybrid aware contention avoidance solution. It also shows that both, BID-Hybrid drastically reduce the total distance traversed by the disk arm as compared to BID-HDD. Similar justification of the amortization effect in BID schemes, as discussed in previous sections, applies for the reduction in disk arm movement distances. Distance traveled is related to the work done, or energy expended, therefore, BID schemes can also result in reduction in the energy footprint of storage systems.

We observe that gains derived by employing BID-Hybrid over BID-HDD is maximum for Tera-Sort (approx. 60%). This is attributed to the nature of the workload having a lot of intermediate non-bulky local disk reads and writes. The large amount of small (in terms of I/O size) operations

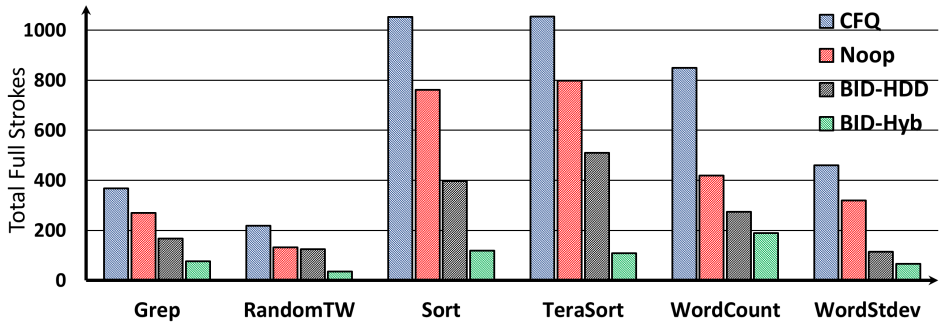


Figure 4.6: Cumulative disk head movement distance.

result in a higher number of disk arm movements, which is spread across the disk layout. Therefore, by removing non-bulky interruption causing I/O requests from the HDD request queue and placing them in SSD, leads to reduction in the overall reduction in average distance per disk arm movement the head has has to move as well as number of disk arm movements, refer Figures 4.5 and 4.6, respectively.

BID schemes (BID-HDD and BID-Hybrid) is aimed to avoid contention following system constraints without compromising SLAs, as described in Section 3.3. Through trace driven simulation and experiments, we shows the effectiveness of both the schemes of BID, i.e. *BID-HDD* and *BID-Hybrid* in shared multi-tenant, multi-tasking Big data cloud deployments. In our experiments, we have shown the impact of block level contention avoidance solutions on single physical storage device (HDD). The effect is additive when applied across all storage devices across the data center. BID essentially increases the efficiency of the block layer by streamlining the serving sequence of I/O requests to the block device in skewed, bulky and multiplexing I/O workloads like MapReduce. Other than resulting in faster I/O completion time, BID schemes can also result in increasing the lifespan expectancy of HDDs, data loss risk mitigation and energy savings due to reduction in the entropy of disk arm.

4.6 Conclusion

We have developed and designed two novel Contention Avoidance storage solutions, collectively known as "BID: Bulk I/O Dispatch" in the Linux block layer, specifically to suit multi-tenant, multi-tasking and skewed shared Big Data deployments. Through trace-driven experiments using in-house developed system simulators and cloud emulating Big Data benchmarks, we show the effectiveness of both of our schemes. *BID-HDD*, which is essentially a block I/O scheduling scheme for disk based storage, results in 28% to 52% lesser time for all I/O requests than the best performing Linux disk schedulers. *BID-Hybrid*, tries to exploit SSDs' superior random performance to further reduce contentions at disk based storage. BID-Hybrid is experimentally shown to be successful in achieving 6% to 23% performance gains over BID-HDD and 33% to 54% over best performing Linux scheduling schemes.

In future, it would be interesting to design a system with BID schemes for block level contention management coupled with self-optimizing block re-organization of BORG Bhadkamkar et al. (2009), adaptive data migration policies of ADLAM [Zhang et al. (2010)], and replication-management of such as Triple-H [Islam et al. (2015)]. This could solve the issue of workload & cost-aware tiering for large scale data-centers experiencing Big Data workloads.

Broader impact of this research would aid Data Centers in achieving their Service Level Agreements (SLAs) as well keeping the Total-Cost of Ownership (TCO) low. Apart from performance improvements of storage systems, the over-all deployment of BID schemes in data centers would also lead to energy footprint reduction and increase in lifespan expectancy of disk based storage devices.

CHAPTER 5. LINEAGE-AWARE DATA MANAGEMENT IN MULTI-TIER SYSTEMS

In the previous two chapters, we have developed and designed storage solutions to manage individuate devices as well as multiple tiers of storage devices from the core of the operating system. We believe that in a large scale shared production cluster, the issues associated due to data management can be mitigated at a way higher layer in the hierarchy of the I/O path, even before requests to data access are made. The current solutions to data management are mostly reactive and/or based on heuristics. In this chapter, we design and develop a data management solution **LDM**, which is both deterministic and pro-active. LDM is designed to cater to a class of applications which exhibit **lineage**, i.e. *in which the current writes are future reads.* In such a class of applications, slow writes significantly hurt the over-all performance of jobs, i.e. current writes determine the fate of next reads. The concepts developed can be extended to a wide variety of applications. LDM amalgamates the information from the entire data center ecosystem, right from the application code, to file system mappings, the compute and storage devices topology, etc. to make oracle-like deterministic data management decisions. With trace-driven experiments, LDM is able to achieve 29% to 52% reduction in over-all data center workload (lineage as well as other concurrent non-lineage applications) execution time.

The chapter is organized as follows. First, we discuss *lineage*, and the problems associated due to over-looking of these class of applications in Section 5.1. In Section 5.2, we discuss the design of our data management framework, **LDM**, and the development of *Block-graphs* required to capture data dependability in workflows. This is followed by designing techniques for life-cycle management (data placement, replica management and tier migration) of data blocks utilizing various tiers of storage. We evaluate the performance of LDM for two types of Data Center lineage-based applications along-with discussions of the results in Section 5.3. The related research

was described in Section 2.1.3. We conclude the chapter in Section 5.4 with a discussion on future work.

5.1 The Problem

Extracting high performance from the storage system is the most important challenge in designing computing systems today. In large data intensive applications, the movement of data from and to storage to compute engine may overshadow the processing time for data [Balasubramonian et al. (2014); Tiwari et al. (2012)]. The storage devices attached directly (or locally) to the compute nodes have limited capacity and are expensive due to their proximity. Therefore, data is typically stored in storage hierarchy and are required to be moved over the network to the compute nodes for processing. Figure 5.1 depicts an illustration where data D are transferred over an I/O channel from storage S to compute node C to be processed by task T. A higher volume of data movement over I/O channels is resource (memory, network, and storage), time and energy intensive. Overall, this scenario makes data storage and management in data centers a challenge as it has direct impact on the efficiency of computing.

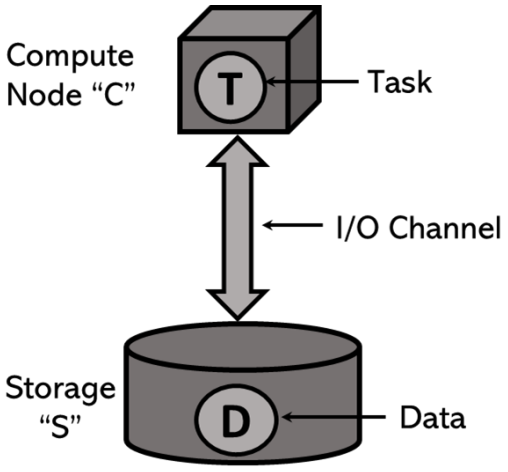


Figure 5.1: Large Data Movement.

Data centers today cater to a wide diaspora of applications which process multiple data sets for multiple jobs in a multi-user environment concurrently. They also deploy storage systems organized

in multiple heterogeneous tiers, which is necessary to achieve cost-performance-capacity trade-off [Mishra and Somani (2017); Iliadis et al. (2015); Kim et al. (2011)]. Dedicating physical resources for every application is not economically feasible. Resource sharing causes contention affecting the efficiency and performance [Mishra and Somani (2017); Mishra et al. (2016); Hindman et al. (2011)]. Despite advanced optimizations applied across the various layers along the odyssey of data access, the data management layer remains volatile [Bjørling et al. (2013); Bhadkamkar et al. (2009)]. Data are scattered over multiple files located at multiple storage nodes[1] and replicated for performance, availability and reliability reasons.

An ideal storage system should deliver the same read (or write) access performance to all applications. The read and write performance may differ due to their perceived implications on the application performance. However, realizing same read (or write) access performance for all applications can be difficult to achieve because the data access time depends on a variety of factors including physical device characteristics, data locations, current utilization of devices, available I/O bandwidth, location of storage device, network topology, and delays, etc.

The current data management techniques fail to capture the syntax and semantics of jobs and the associations of data in various stages of jobs. Moreover, the goals of current efforts have been to make read operations faster as they are believed to be the biggest bottleneck. However, inconsiderate placement of intermediate results (writes) for reuse may affect the read performance adversely. Under this scenario, the gains derived by deploying multiple tiers in storage can be nullified easily by improper replica allocations to tiers, handling of memory resources, and avoidable data movement [Iliadis et al. (2015); Zaharia et al. (2012); Li et al. (2014)].

Here, *we address the issue of how to deliver ideal system performance for all read and write accesses.*

We understand that it can be and is hard to manage data storage and movement for any arbitrary set of applications contributing to data center workloads. Therefore, to begin with, we limit our scope to a set of applications that depict the lineage property, where intermediate data

[1]Storage refers to the overall data plane, whereas a storage node refers to a single physical device.

during computation must be written and read back later on. Such a scenario occurs commonly in data centers. For example, in one application scenario, data may be extracted using MapReduce [Dean and Ghemawat (2008)] which are queried using Pig [Mishra et al. (2017)], then machine learning algorithms are used on the queried results [Li et al. (2014)], and are finally combined with other similar results to produce the final answers [Li et al. (2014); Bu et al. (2010)]. The issues associated with data management gets amplified for applications with such chained jobs, which exhibit lineage. It is now becoming clearer that dealing with large amounts of current "writes", which are future "reads" is equally important to achieve good performance [Li et al. (2014); Ananthanarayanan et al. (2012)]. Multi-tier storage offers multiple dimensions, such as device type, network connectivity, and replication management, allowing exploration of data access solution space differently.

We develop and design a novel framework, called **LDM**, to address the challenges in lineage-aware data management to effectively utilize multi-tier storage hierarchy. LDM captures the inherent lineage information and reduce the data movement via network by placing them appropriately to enable maximal processing nearer to the storage locations as well as in appropriate storage tiers. Moreover, LDM utilizes all tiers[2] of storage to reduce data access delays in conjunction with workload aware tiering[3] by orchestrating multiple data management features. These include data placement, data replication management and data migration. We believe LDM will have a huge impact on the performance and resource management of data processing platforms. From the Green Computing perspective, our solution will decrease energy footprint, due to much reduced work to process data across all tiers of computing, i.e. storage, compute (required on storage servers), and network.

[2]Storage media across all nodes with similar I/O characteristics form a tier.

[3]Tiering refers to orchestrating data between heterogeneous tiers of storage by leveraging individual strengths of each to maintain balance between Cost, Performance and Capacity.

5.1.1 Motivation

The following trends are clear:

1. The size of data is ever increasing, and more and more data are being stored on remote storage.

2. The complexity and structure of data being processed for analytics varies dramatically.

3. Enterprise data centers includes thousands of processing (and heterogeneous) and storage nodes.

4. Memory needs for data processing is increasing.

5. Data is organized in multiple heterogeneous tiers with a wide variety of storage devices in both local and remote storage to extract best performance.

6. Increasing amount of data are being used by multiple applications and/or series of jobs of the same application chained together.

7. Chained MapReduce ETL pipelines and Oozie workflows are most popular lineage based applications.

8. Current writes are future reads. Thus, writes dominate the over-all performance of these applications. Therefore, more emphasis needs to be given to initial data placement and replica management.

9. Data management needs to be both ecosystem as well as network-storage architecture-aware.

Most studies have focused on studying data center operations to consolidate the computing needs and organize and optimize computing for multiple applications. Computing resources are believed to be abundant, but without appropriate attention, they are mostly waiting for data and wasting cycles [Mishra and Somani (2017); Bjørling et al. (2013); Bhadkamkar et al. (2009); Choi et al. (2017); Bu et al. (2010); Afrati and Ullman (2010)]. Moreover, for lineage based applications, the impact is more severe due to data-dependency between tasks. Keeping all the data in memory (as done in Spark) may not be a wise choice either. We believe that the focus needs to shift from computing to data. What makes this shift relevant is the availability of oracle-like deterministic workload and data center storage topology aware data management. Datum access from storage and copying in memory is expensive. Therefore, we believe that studying data utilization patterns

and developing strategies to optimize computing paths are the greatest needs at the current time [Zaharia et al. (2010)].

5.1.2 Broader Impact and Goals of LDM

LDM fulfills the need of effective utilization of storage tiers and develop a solution framework to enable lineage aware data management. Specifically, we will consider disks based storage and storage class memories (SCMs) as a set of conjoined resources to form a storage continuum. We design LDM as an elastic Lineage-aware data management framework to work with a class of chained data processing applications on a wide variety of storage hardware. LDM will work on stand alone systems, on dedicated clusters, and on cloud based data centres. LDM is designed to be elastic and be able to adapt to the current resource availabilities in a data processing cluster.

Another goal of LDM is to place data based on the current resource availability and application performance requirements such that the computation will find them closer (time) to it. Depending on the current state of the system the adaptive data management features like initial data placement, replica management and migration policies can be used to organize data on disks, solid-state devices either locally on computing servers DAS (direct attached storage) or in remote NAS/SAN (network attached storage or storage area network) arrays. LDM will enhance the software-defined storage capability by providing a holistic approach of managing data-task associations by creating flexible tiering.

We develop methodologies to improve the effectiveness of resource allocation. Our goal here is to capture the inherent lineage information of chained jobs of the same application that have data dependency, manage current writes, utilize data placement for next reads, and multi-tier (storage device as well as network-storage architecture aware) resource allocations that exploit data locality and parallelism. The goal is to demonstrate that it is possible to maximize resource utilization while addressing the needs of individual application performance. In our research, the focus is shifted to orchestrate the application requirements what storage delivers to what the storage is capable of delivering. Moreover, deploying extensive pre-processing to create efficient data consumption

pipelines will reduce the write and read delays, and increase the efficiency of data processing clusters.

LDM will allow resource managers to reduce the need for computing and network resources. They also can fine tune computations based on the application performance requirements and the available resources.

5.2 LDM

In large scale distributed systems, data management plays a vital role in processing and storing primary and backups of data across storage devices. Despite advanced optimizations being applied across various layers along the odyssey of data access, the data management layer still remains volatile [Bhadkamkar et al. (2009); Bjørling et al. (2013)]. Goals of current efforts are to make read operations faster as they are believed as the biggest bottleneck. However, inconsiderate placement of intermediate results for reuse may affect performance adversely. This problem gets amplified for chained applications which exhibit lineage. It is now becoming clearer that dealing with large amounts of current "writes", which are future "reads" is equally important to achieve good performance. Therefore, in such data processing pipelines, its imperative to capture lineage or relationship across tasks and their dependency with data, i.e. data-task associations.

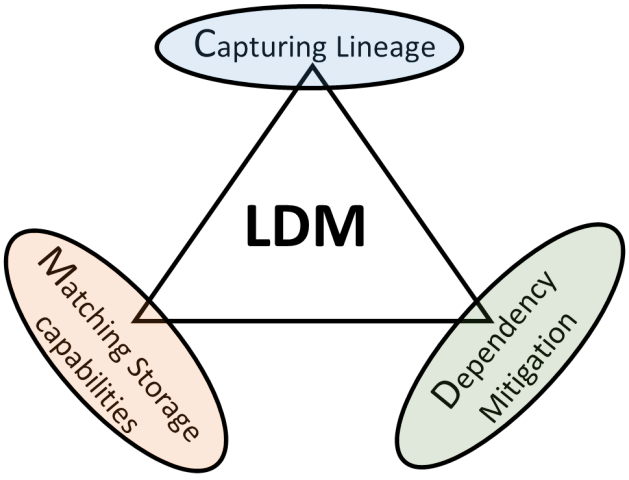


Figure 5.2: Components of LDM.

There have been efforts [Li et al. (2014); Zaharia et al. (2012)] to understand lineage for in-memory computation for improving job recovery time in-case of fail-overs and performance in Data Centers with nodes having large memory. Zaharia et al. (2012) forms distributed data-sets for in-memory computations (production and computation in-memory), which inherently improves performance. Li et al. (2014) proposes an in-memory fault tolerant mechanism which leverages lineage to recover lost outputs by re-executing the steps which formed the data-sets. In-memory computations and storing of results in memory are infeasible for Big Data workloads as the working sets are huge to fit in RAM, along-with the time-varying nature of applications for production and consumption of data blocks [Harter et al. (2014)]. Issues such as ensuring reliability and cost-effectiveness are other major challenges in such frameworks. Therefore, cost simulations in Harter et al. (2014) that adding small SCM tier and efficient orchestrating data between tiers can lead to enhanced performance than equivalent spending on RAM or disks. Multi-tier storage offers multiple dimensions, such as device type, network connectivity, and replication management, which allows to explore to explore the issues associated with data access differently.

Multiple solutions [Kakoulli and Herodotou (2017); Grund et al. (2010); Islam et al. (2016); Krish et al. (2014b); Lee et al. (2016); Gunda et al. (2010); Olson et al. (2017); Zhang et al. (2010); Mihailescu et al. (2012); Ananthanarayanan et al. (2012); Iliadis et al. (2015); Islam et al. (2015); Grund et al. (2010)] have been proposed in literature to exploit multi-tier storage, but none addresses the issues associated with lineage or chained jobs. The storage layer is agnostic to the semantics of tasks on data and its execution characteristics. Our goal is to explore the inherent lineage information (data-task associations) coupled with multi-tier storage for chained job class of applications.

In LDM, we provide a uniform execution environment across the storage server and compute server. We address the specific needs of a cluster of applications with data-dependencies. LDM resides in the Master (or Head) node of clusters where jobs are submitted by applications and data management decisions are made. LDM includes the following three components.

1) Capturing Lineage Information. This component is accomplished by a set of APIs that generate and analyze the metadata associated with application code and extract semantic knowledge of the computational workflow and logic from the code to build task and block graphs (described later).

2) Matching Storage capabilities. This component uses the information about the storage devices (type, capacity, performance, etc.) as well as location (local or remote) to categorize and classify them. These APIs reside on DataNodes and storage servers and transmit storage devices information as part of the status updates regularly to the NameNode for its own use in making data location decisions.

3) Dependency Mitigation. This component use the above two modules assisted by the information stored in the Distributed file system to make data management decision and policies. These include APIs for initial Data Placement and Replica Management to decide where to place the date and its copies in terms of tier and device. Data Migration API will use the lineage information to evict already placed blocks as well as determines the utility of the blocks for both capacity and efficient utilization of storage tiers.

Before describing these components in detail, we first describe Hadoop and MapReduce ecosystem in brief as this will be used as a running example to demonstrate the need and our approach to solution.

Hadoop Ecosystem and MapReduce

Hadoop and its data processing framework - MapReduce is the de-facto large data processing framework for Big Data. Hadoop is a multi-tasking system which can process multiple data sets for multiple jobs in a multi-user environment across multiple machines at the same time [Krish et al. (2016); Vavilapalli et al. (2013)]. Each MapReduce job consists of multiple processes submitting I/Os concurrently for Map, Shuffle and Reduce stages, each having skewed I/O requirements [Lu et al. (2017); Ananthanarayanan et al. (2011); Moon et al. (2015)]. Hadoop Distributed File System (HDFS) uses a block-structured file system to deliver reliable storage [Mishra et al. (2017); Vavilapalli et al. (2013)].

YARN (Yet Another Resource Negotiator) is used for per-application based resource negotiating agent and is a centralized platform to ensure consistency and data manageability. YARN has enabled Hadoop with the flexibility to encompass multiple data processing engines such as Spark, Storm, etc. to process and manage the data concurrently. It has also enabled Hadoop with multi-tenant processing capabilities such as different applications/ processing engines working concurrently by using application based containers.

HDFS splits the files into fixed size file system blocks (64/128 MB), known as chunks, which is typically tri-replicated for achieving the fault-tolerance, availability and performance parameters. HDFS follows a leader-follower architecture, with a NameNode, which manages storage and several DataNodes hosting the data [Vavilapalli et al. (2013); Mishra et al. (2017)]. he NameNode manages the file system namespace and associated metadata (file-to-chunk maps) as well as contracts the access to files by clients (once brokered, the clients interact directly with DataNodes). The NameNode operates entirely in memory, persisting its state to disk. All such information are persisted in two major files in the NameNode: 1) *fsimage*, which stores the complete snapshot of the file system metadata at a particular instant; and, 2) *edit log* for the incremental changes to the file system namespace [Vavilapalli et al. (2013)]. Thus, the NameNode is central to data management and can be exploited in developing necessary policies.

5.2.1 LDM framework

The key idea behind LDM is to build a data management system which has an oracle-like capability to know the future usage of data and therefore take deterministic actions based on its knowledge. The information of the data processing framework and ecosystem is already present and needs to be properly harnessed. All such information and statistics are already being produced or consumed by different components of the system. Therefore, this knowledge when amalgamated with our tier-and-network aware storage policies should yield better performance for lineage class of applications.

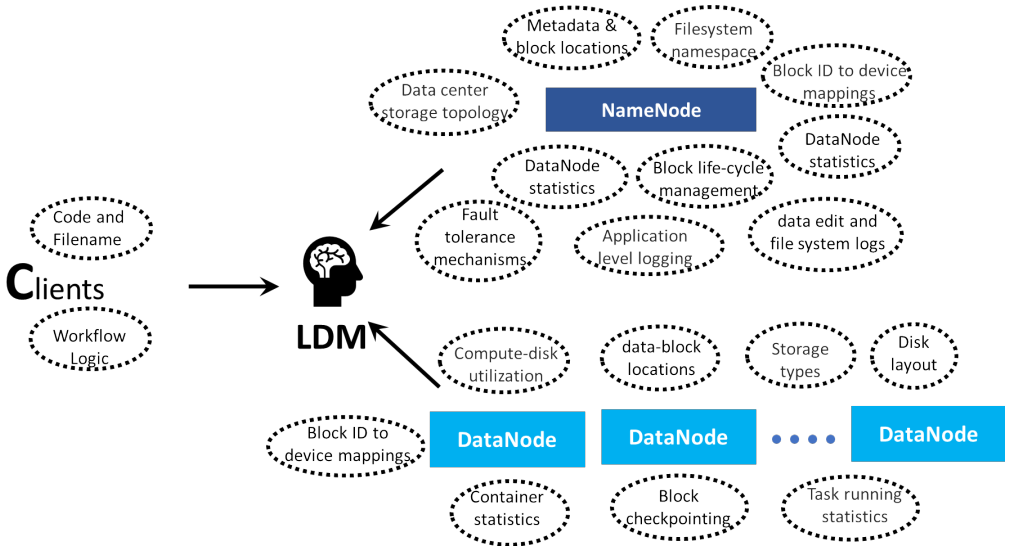


Figure 5.3: LDM Knowledge mining to aid data management policies.

In Hadoop like environments, LDM would work with YARN resource manager and HDFS data management to transform information into intelligence and use the intelligence acquired to execute policies for to mitigate the impact of data dependency for lineage class of applications. Consider Figure 5.3, which briefly describe the information produced by different system components. LDM can use this information to create data management policies to achieve better performance for applications exhibiting lineage.

In the following three sections, we describe the details of the working components of LDM.

Capturing Lineage Information

It is imperative to understand and extract information about the workflow and the dependencies within and among the applications. The semantics and syntax of every application can be extracted by mining the code and amalgamated with the data processing steps to understand the ecosystem and achieve efficiency in the computation. We achieve this functionality by developing a set of APIs. The Client API will understand the computation flow and build task graphs based by mining the application code. A task graph will be represented by a DAG (directed acyclic graph) representing the tasks as nodes and associated dependencies between tasks as directed edges.

A task graph alone does not exhibit the location based data dependency. LDM, therefore, will use the file-system information about the block to device mappings (for example: fsimage, and editlog for HDFS) to associate blocks of data with tasks (using task blocks) to develop ***Block graphs***. Block graphs capture all the data block-task associations, which would deterministically capture data lineage across tasks. This knowledge would aid in mitigating the impact of delays associated to writing and then subsequently reading intermediate results. The interaction between the Client API for task graphs and filesystem namespace is achieved by the Data block-Task Associativity API working on the NameNode.

Understanding Ecosystem: Job Pipelines

Chained Jobs are a popular class of applications that are executed on clusters. Essentially, the jobs are pipelined and the output of a job forms the input (or a part of the input) of the next job. Such jobs are common in several business and scientific applications. For example, Job pipelines are produced by Hadoop workflow managers like Oozie to perform ETL (Extract, Transform and Load) applications [Li et al. (2014); Kakoulli and Herodotou (2017)]. Data is extracted using MapReduce, then queried using Pig, followed by machine learning algorithms delivering the query results, that are combined with other results [Li et al. (2014)]. Clearly, there is data-dependency between jobs.

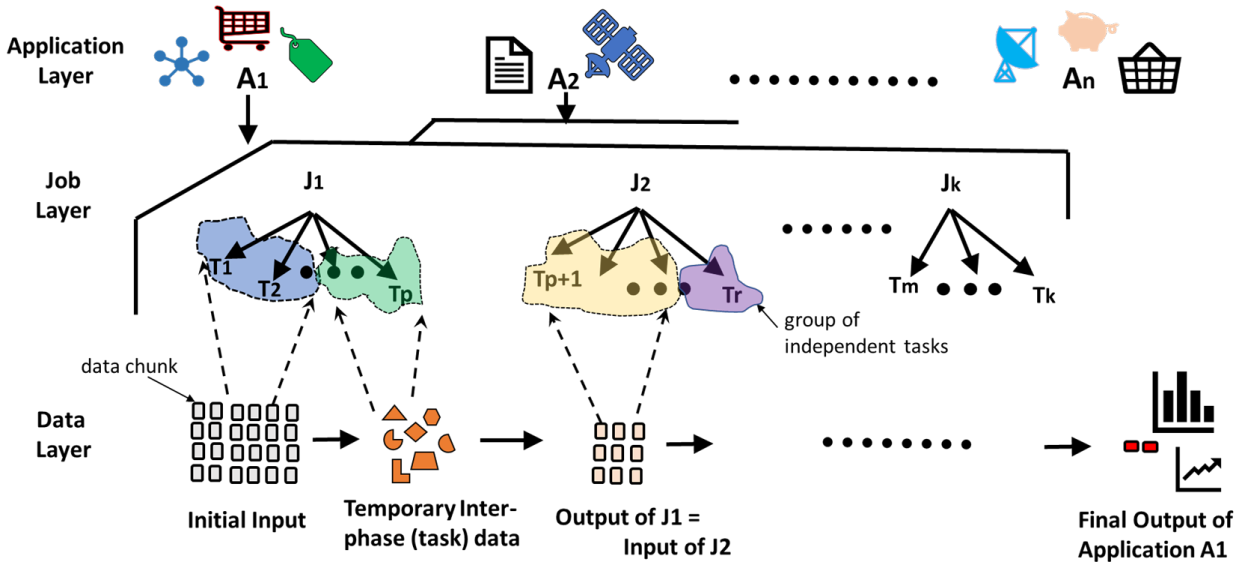


Figure 5.4: Job pipelining and data-task associations.

A typical data center workload scenario consists of multiple applications having highly skewed I/O characteristics that exists concurrently as concurrently as shown in Figure 5.4. The following hierarchy commonly is maintained for data processing.

1. Application Layer: Data Centers spawn multiple instances of many applications using compute and storage servers. As depicted in Figure 5.4, n applications A = $\{A_1, A_2, \ldots, A_n\}$ are waiting to be executed on the cluster at some time instance 't'. These applications vary in nature (i.e. analyzing customer behavior, weather patterns, genomics, financial data, etc.) and are independent.

2. Job Layer: An application A_i, is a conglomeration of several jobs denoted by set J = $\{J_1, \ldots, J_k\}$ with data-dependency. In Figure 3, application A1 is shown in detail as consisting of jobs J_1 to J_k.

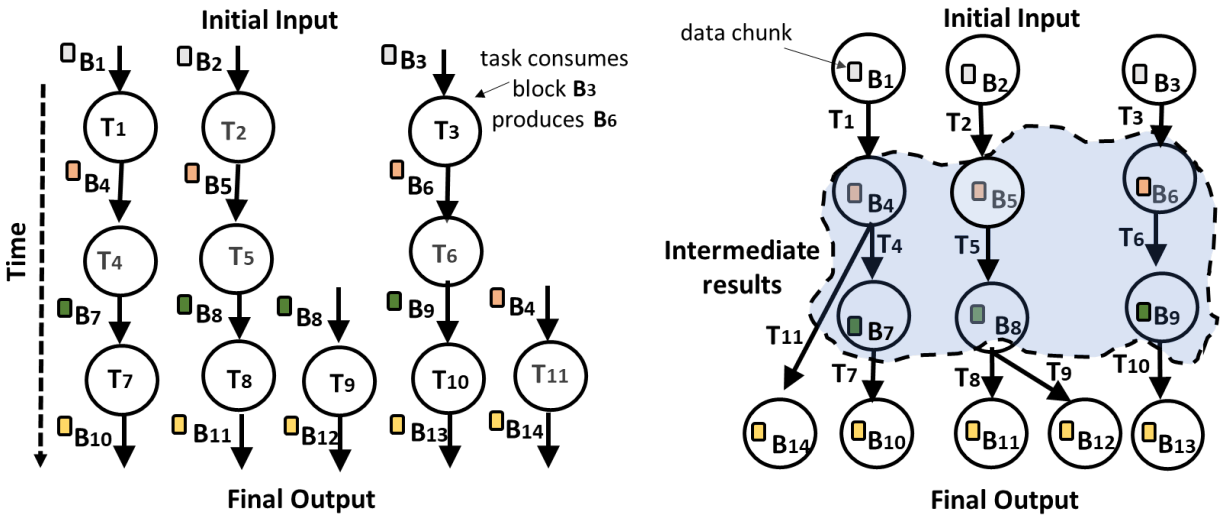
3. Task Layer: Task is the smallest granularity of computation which access and process data. Each job J_i in an application is further sub-divided into a set of tasks T = $\{T_1, \ldots, T_p\}$. In a typical scenario, multiples such tasks are run concurrently. Depending on the application, the intermediate results may be required immediately as well as later on.

Data Layer: A task consumes or produces data which is organized across multiple heterogeneous tiers of storage. Due to storage virtualizations, multiple processes contend for the same physical resource for read and write access that may be stored in local (DAS) or in remote storage (NAS/SAN). I/O across the network exposes data transfer to network congestion, delay and losses.

Task graphs

A task graph is a sequence of all the tasks associated with each other and is represented by a DAG (directed acyclic graph). Task graphs can be formed by mining of the application code coupled with logic extraction of the ecosystem. Based on the framework like MapReduce, Spark or Pig, and the logical data flow, the NameNode can create the task graph when an application is submitted. The task graph provides an overview of job and dependencies among tasks currently running on the system as well as maps the application requirements. Figure 5.5a shows the task

graph for a set of inter-linked or chained jobs.



a) Task graphs-workflow

b) Block graphs: Data-task associations

Figure 5.5: Dataflow representations (a) Task graphs- workflow; (b) Block graphs: Data-task associations.

Block graphs: Data-Task associations

Despite capturing the workflow, a task graph fails to understand the lineage of data and associativity of different tasks and blocks of data as conjoint pairs. This is extremely fatal, as a data block might be consumed/produced by multiple tasks and affect the over-all performance of the application. There is a clear need to extract these relationships and utility of each data block to develop proper data management policies. LDM uses the file-system information about the blocks to device mappings (similar to fsimage and editlog for HDFS) stored in the NameNode to associate blocks with tasks to construct a Block graph. In LDM, the Data-Task Associativity API working on the NameNode captures these interactions. The knowledge of these interactions aid in mitigating the impact of delays associated to writing and then subsequently reading intermediate results.

Block graphs are essentially representation of data-task associations, where the blocks of data (as a single entity or replicas) form the vertices and the tasks producing/consuming them as edges as shown in Figure 5.5b. The utility and reusability of blocks can be determined using this repre-

sentation. During run-time, only initial input data (initial filename) is available, the initial vertices (block-IDs) are formed using the filesystem namespace and the logical tasks. The graph for later stage is constructed assuming every task produces a data block (not necessarily a full data chunk). Application code mining can help determine tasks consuming/producing a data block, thus providing a complete overview of data usage for the computation.

An Example: A MapReduce job: DAG of tasks and Block Usage

We use a MapReduce application source code analysis to describe how the Client API can extract the entire computation logic (task graphs) and the Data-Task Association API integrates the filesystem namespace to build block graphs. Figure 5.6 depicts a simple MapReduce function to count the number of instances of unique words in a file of multiple TBs in size. The code is broken into simple Unix commands, such as, cat, grep, and wc -l (subtasks), and these logical partitions may be executed in different machines.

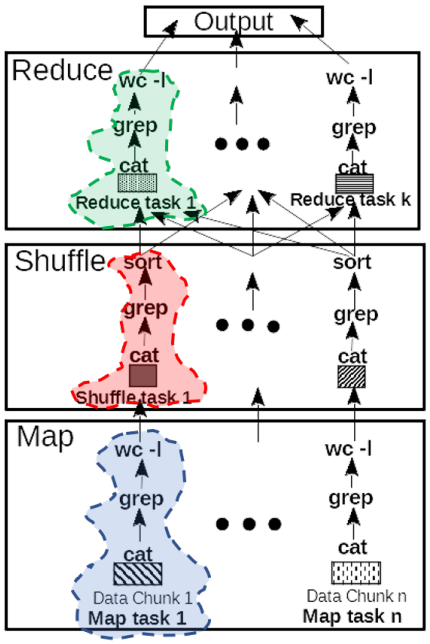


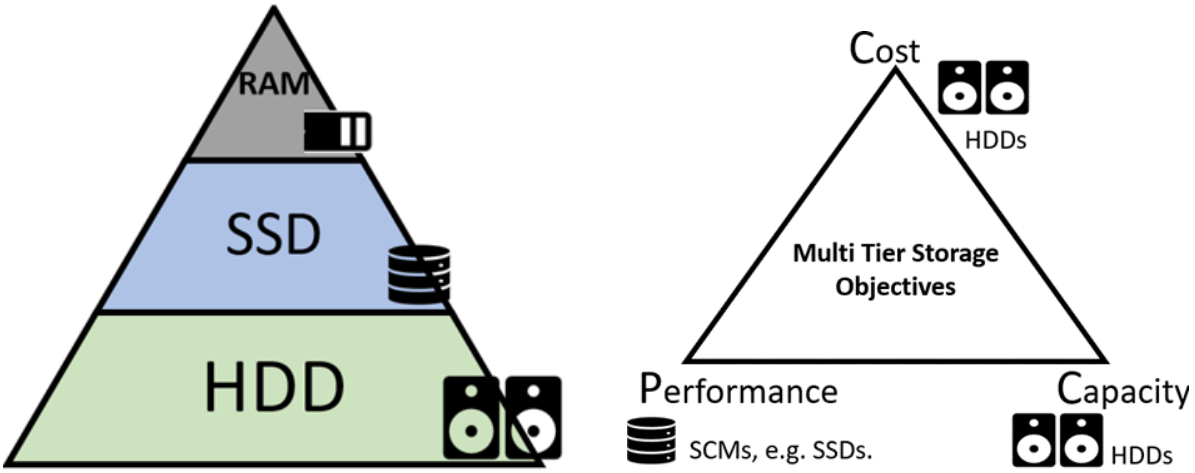
Figure 5.6: MapReduce Job: DAGs of tasks.

In the Map phase, 128 MB data chunks are read from storage nodes and are processed to form collection of $< key, value >$ pairs. The Shuffle stage partitions the output of Maps based on "keys"

to be processed and transfers data to reducers by the Reduce tasks. The DAG structure and data needs data for computation to be persisted in storage. *Figure 5.6 clearly depicts the lineage, task graph and how a block graph can be generated from it.*

5.2.2 Matching Storage capabilities

Hadoop like large distributed systems gained popularity due to their design of bringing computation closer to data which were primarily for DAS setups of acomprising large number of inexpensive machines. However, as we move forward and data being scattered, it is necessary to deploy remote storage servers (NAS/SAN) across thousands and millions of storage devices with varied I/O and physical characteristic. The storage hierarchy include Main Memory (RAM), Solid State Drives SSDs, and Hard Disk Drives (HDDs), as shown in Figure 5.7a. Storage media across all nodes with similar I/O characteristics form a **tier** [Iliadis et al. (2015); Li et al. (2014); Kakoulli and Herodotou (2017)]. The question of when, where and how-to organize data over multiple tiers in the hierarchy becomes important to reap the maximum benefits.



a) Memory Hierarchy b) Multi-tier storage objectives

Figure 5.7: Multi-tier (a) Storage Hierarchy: descending order in performance and cost, and ascending in capacity (top-down); (b) Storage design objectives.

Software defined storage (SDS) is part of the solution to deliver storage services. Automated tiering is one of the major focus to deliver SDS [Di Mauro et al. (2017)]. Recall that *Tiering* refers to orchestrating data between heterogeneous tiers of storage by leveraging individual strengths of each to maintain balance between Cost, Performance and Capacity, as shown in Figure 5.7b. Our goal in LDM is to use the information about the storage devices (type, capacity, performance, etc.) as well as location (local/remote) to categorize and classify them. The next step is to implement appropriate APIs to reside both at the DataNodes and storage servers to send their usage status regularly to the NameNode. The NameNode will use the information in making data location decisions.

Need for multiple tiers and automated tiering

Disk-based storage devices are the backbone of data centre storage. HDDs provide the perfect blend of cost and capacity to satisfy the volume requirement of Big Data. But due to their physical limitations, non-volatile devices, also known as storage class memories (SCMs), such as SSDs are also being used in large data centres. SCMs offer superior access time due to non-moving parts. SCMs used with legacy disk based interface such as SATA/SAS were incapable of harnessing the throughput and inherent parallelism. However, recent advances such as faster PCIe bus technology (also known as NVMe Express) [Lee et al. (2017); Malladi et al. (2016)], PCIe switches, Linux block layer redesign, etc. are enabling SCMs to provide higher performance.

Despite superior random performance of SCMs (or SSDs) over HDDs, their higher costs, need for write amplification, and lower lifespan remain concerns for long-term economically feasibility. A hybrid approach with heterogeneous tiers of storage such as those having HDDs and SCMs coupled with workload aware tiering to balance cost, performance and capacity have become increasingly popular [Khasnabish et al. (2017)].

Existing definitions of tiers considers only device characteristics. They do not take into consideration proximity to compute resources and effects of network transfers. Data in a local HDD might be more valuable than in a SSD in a remote location. However, it is not possible to store all data in local storage due to the working set sizes, large spectrum of concurrent applications running

on a node and their varying I/O characteristics. Remote storage offers an alternative to satisfy the volume requirement. However, performing I/O across the network makes data transfer prone to network issues like congestion, delay and losses. Such data movements are thus expensive and affect application performance. Therefore, resource manager typically monitors network congestions and tries to utilize replication to maintain performance.

An intelligent and deterministic data management technique would orchestrate the application needs apriori to minimize data movements leveraging ecosystem tools (replication, tiers, etc.) efficiently while ensuring performance. An ideal data management scheme should envision local-remote storage conjoint-pairs analogous to in a similar manner as the cache-main memory model, but with a different interface.

Replication: Use It as a tool

The data chunks are usually replicated and stored in different nodes across the storage system. The purpose of replication is tri-fold, i.e. achieving fault-tolerance, availability and performance (consider Figure 5.8). The current schemes [Krish et al. (2016); Islam et al. (2015); Spivak et al. (2017); Khasnabish et al. (2017); Krish et al. (2014b)] of replica management leverage the number of copies or replicas and their locations. Islam et al. (2015) designs a heterogeneous storage engine for HPC including RAMdisks, SSDs and HDDs, and Lustre FS to benefit HDFS. The data placement engine in Islam et al. (2015) deals with tri-replication of blocks to ensure fault tolerance and the decisions of placement of replicas in a tier is based on storage space available with a usage-priority based tier migration model. Krish et al. (2014b) proposes a model with an intent to remove performance bottlenecks by placing every block belonging to file in all tiers of storage.

An interesting approach to replica management would be to i) take into consideration the tier-device characteristics and device utilization and ii) move replicas dynamically based on time-varying application I/O requirement. This adds another dimension which would be highly beneficial to all applications, especially, those which have data-dependency, i.e. lineage based applications. Currently, tiering is mostly concerned with defining hotness or randomness and not on replica man-

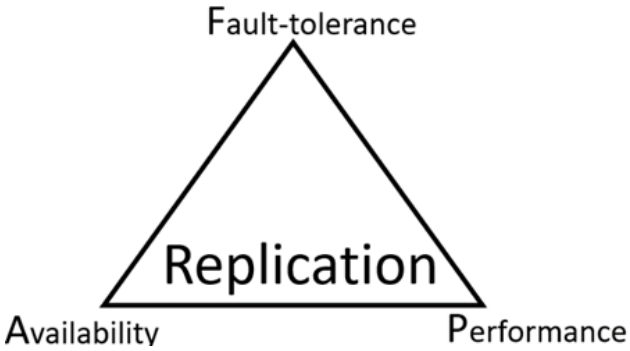


Figure 5.8: Advantages of replication.

agement as a tool for multi-tier environments, as discussed in Chapter 4. Well-managed multiple replicas of data will lead to better performance.

Storage Device Classification and Categorization

The classification and categorization of storage devices refers to associating them with a *performance-score or rating, PR*. *PR* includes performance governing parameters like speed, remaining capacity, number of channels for I/O access (i.e., one for SATA/SAS HDD or SSDs, 8/16 for NVMe SSDs), current utilization, location in storage architecture (DAS or NAS/SAN) and cost/GB. The key goal behind PR is to overcome the deficiencies of current practices and integrate them into the data management policies and tools.

PR is a dynamic parameter for each device as the factor defining them vary over time. The appropriate APIs on DataNodes and the storage servers can collect all such relevant parameters for the devices attached to them, and periodically transmit them to the NameNode via status updates. The Storage Classification API residing on the NameNode utilizes all these parameters to profile every storage device at the run time. The value of PR for a storage device and the location of the client requesting write/read for data placement allows a correct decision to be made. For example, the value of an HDD attached locally can be higher than a SSD in a remote location (location w.r.t. computation locality), or vice-versa. Such decisions are complex and tiering decisions cannot be made solely on device characteristics. Therefore, LDM provides a dynamic and unified method with the use of performance ratings to profile the available resources to aid data management decisions.

5.2.3 Dependency Mitigation

Our goal is to focus on mitigating the impact of associated delays due to incorrect management of data dependency for applications exhibiting lineage. The knowledge gained by using the task and block graphs and the awareness of the storage systems capabilities allow us to develop sound data management policies. Primarily, we will perform *Initial Data Placement, Replication Placement,* and *Data Migration* tasks to decide the storage device(s) to place the data and if, when, and where to move data blocks dynamically.

The Dependency Mitigation component uses the lineage information and the PR values to deliver data placement and replica placement decisions. It includes two APIs, one for initial Data Placement and one for Replica Placement. Data Migration API uses the lineage information to evict already placed blocks as well as determines the utility of the blocks for both capacity and efficient utilization of storage tiers. *Please note that LDM is concerned only with the data management of intermediate results. For some tasks, initial data can be treated as intermediate results if the data is being migrated for the computation.*

HDFS Data Placement: An Example

Currently in HDFS, during the write phase, as soon as a worker writes 80% of a data chunk (64 MB) in memory, it tries to persist the data chunk in storage [Vavilapalli et al. (2013)]. The worker contacts the NameNode through RPC calls for a list of DataNodes which can host the chunks and its copies. The NameNode follows rack-aware fault-tolerant algorithms to protect against network failures for the placement of data, generates a list of available DataNodes to the client. The client directly communicates with all DataNodes and pipelines the data chunks one-by-one in 4KB data packets following an ACK based protocol received [Vavilapalli et al. (2013)]. This pipelining slows the write process, as to maintain fault tolerance and consistency the slowest DataNode governs the over-all performance. The current schemes do not leverage the tiers of storage available for placement of initial data and its replicas. We propose to use a different strategy in LDM as outlined below.

ALGORITHM 4: LDM Initial Data Placement

for *every application* $a \in A$ **do**
  Compute block-graphs $BG_a \in BG$;
for *every block* $B \in BG$ **do**
  Compute LQ_B using reusability R;
  Refer global storage table GST to compute refactored PRs
  (based on locality L of task producing block B);
  Select storage media m based on LQ_B and refactored PR;
  Send location of storage media m to Worker;
  Pipeline replicas as per Algorithm 5.

Initial Data Placement

With the knowledge of all tiers in storage media (PR), the current write as well as retrieval needs, and the knowledge of data-task associations of all the tasks currently running or to be spawned in near futures with the help of the block graphs, LDM is better equipped to dictate policies for placing the initial data across the storage to benefit future reads. When a worker contacts the NameNode for writing a data chunk B, the Initial Data Placement API residing on the NameNode uses the block graphs to compute the data-dependency factor, known as "*Lineage Quotient* LQ_B" for the data block B. LQ_B determines the utility value of block B and based on it, a storage media is selected where the data should be initially placed. To unify the placement algorithm, a global storage table (**GST**) for placement is maintained. GST suggests the storage medias (based on recently computed PRs) that are suitable for a range of LQ_B as described below.

Algorithm 4 describes the working of the Data Placement API. LQ_B is determined by investigating the *reusability* R, i.e. the outdegree of block B from the block graph. Lineage factor LQ_B increases with the number of tasks using it next, i.e. (*reusability* R). The closer the data is placed to the task generating it, lesser is the network footprint usage. Therefore, *locality* L plays a vital role in determining the appropriate rack and storage media in it for the initial placement of data. For all the storage devices, the performance rating PR is refactored based on network distance from *locality* L, higher the distance lower the performance rating. For example, for a task generating block B, a HDD with lower network distance might have a higher refactored performance rating

than a SSD which is far away. Based on the *global table GST*, storage media and its location is sent to the worker to write the data.

ALGORITHM 5: LDM Replica Placement

for *every replica* r *of block* B **do**
Compute Reusability factor of replica r, $R^r = (R-1), R^r \geq 0$;
Compute LQ_r using reusability R^r;
Refer global storage table GST to compute
refactored PRs (based on locality L of previous replica
and storage rack-aware policy);
Select storage media m based on LQ_r and refactored PR;
Send location of storage media m to Worker for pipelining replica;
$R = R^r$, previous replicas reusability factor.

Replica Placement

Once the first data block (4KB) of the 64MB data chunk is written to the storage media, the replicas needs to be placed in the pipeline. The principle is similar to the current fault tolerance mechanisms of the Hadoop ecosystem. The placement of replicas is managed by the Replica Management API residing in the NameNode. The current schemes, take into consideration fault tolerance for placement of replicas to guard against network failures. To mitigate dependencies, the write throughput of all the pipelined replicas is important which is governed by network bandwidth and storage media. LDM manages this as follows.

Algorithm 5 describes the working of the Replica Placement API for pipelining replicas. Similar to initial data placement the Lineage Quotient for (two additional copies for tri-replication) blocks LQ'_B and LQ''_B, respectively, is calculated with additional parameters. The reusability factor R for every replica is reduced by 1 (with minimum 0) from the previous replica value. Here, the locality L of the previous replica is used only to determine a separate rack for storage than the initial block to respect fault tolerance. The assumption here is to use replicas for satisfying the performance for parallel tasks trying to consume the same data set as well as effective capacity utilization of tiers. Therefore, the probability of all replicas occupying the fastest tier is reduced, unless it is a highly dependent block. Based on the Lineage factor of the replicas, the locations are determined in a similar manner to the placement of initial data. The worker is informed of the storage media and the locations for both the blocks and it follows the data write pipeline ACK protocol.

ALGORITHM 6: LDM Data Migration

for *every block (and their replicas)* $B \in BG$ **do**
  if *event time* t *has passed* **then**
    Reset Recency list R_l^B and Recency R^B;
    Compute MQ_B using R_l^B and R^B;
    Match MQ_B with global storage table GST
    to compute appropriate storage media m;
    if *Tiering decision is made* **then**
      Tier Block B to storage media m

Data Migration

Once, the data and its copies are placed, it is imperative to move data blocks (as well as copies) between storage devices (inter or intra tiers) to achieve cost-performance-and-capacity trade-off. The fast tiers are expensive, and the capacity is limited. Therefore, they should be used wisely with most performance critical blocks in them. The Data Migration API uses the lineage information (block graphs) to determine the dynamic time-based utility value of the blocks which is used to dictate data migration policies.

The *Lineage Quotient* LQ, of the blocks (and its replicas) decides the placement of data to appropriate tiers, and it would be economically infeasible to store the data in the same media forever even when their utility is over. LDM will maintain a dynamic and time bound *Migration Quotient* MQ_B for every data block B. MQ is similar to Lineage quotient LQ except that MQ integrates a time bound factor with the reusability.

Algorithm 6 describes the working of the Data Migration API. The Data Migration API maintains a list of event-based time between reuse of the same data block, known as *Recency list* R_l. *Recency* R_c is the time to next reuse. Once the event-time t has passed, the *Recency value* R_c is refreshed. This can be accurately determined with the help of block graphs as it opens an avenue to foresee the data-task association and its usage pattern. Recency R_c captures the time based utility of data blocks, which can also be referred to as deterministically computing the temperature of data. Therefore, MQ can be used to determine the "actual" hotness of data and therefore aid in data eviction or promotion.

Data Migration API is triggered at fixed intervals of time, where MQs of all the already placed blocks (and its replicas) is calculated and tiering decisions are made. MQ values of all the blocks are matched in the global storage table GST and accordingly the block is tiered to the appropriate storage device. This is contrary to the existing data migration policies, where data is evicted or promoted based on its previous history of usage. They are based on trial-and-error and on the assumption that data which is hot in the past has high probability to be hot at a later time. LDM brings forth a deterministic method for life-cycle management (movement) of data and its replicas.

5.3 Experiments and Performance Evaluation

Through trace-driven and log-based simulations, we evaluate the performance of LDM and compare it with the current implementation of YARN (and HDFS) using our in-house developed system simulators. We discuss the testbed setup and performance evaluation in this section below.

5.3.1 Testbed setup

Our experimental testbed consist of our Hadoop cluster and trace (and log) collection remote nodes. The Hadoop cluster topology consists of 1 NameNode, 1 Secondary NameNode and 8 DataNodes. Each node has 16 cores (Two 2.0 GHz 8-Core Intel E5 2650), 128 GB of memory, GigE and QDR (40Gbit) Infiniband interconnects, and 2.5 TB Hitachi HDDs. We use CDH v5.11.1 (Cloudera Hadoop) with the latest implementation of YARN and HDFS. The heterogeneous capability is achieved by using specifications similar to 256 GB Samsung SSD 840 pro with the distribution of capacity in the ratio of 1:8 as compared to HDDs attached locally to the nodes.

We select industry and academia wide used Hadoop benchmarks considering a wide diaspora of I/O workload characteristics, as specified in HiBench [Huang et al. (2010)] & TPC Express Benchmark (TPCx-HS)- Hadoop suite [TPC(tm) (2016)]. These benchmarks have been designed to recreate enterprise Big Data Hadoop cloud environments, stressing the hardware and software resources (storage, network and compute) as observed in production environment. We use benchmarks to form long-running lineage applications which have inter and intra job data dependency.

We also run non-lineage concurrent applications to emulate a realistic shared big data infrastructure. We discuss the details of the applications in the next section.

The NameNode and DataNode statistics, job (and container) running history along with File System details (block locations, chunk-to-device mappings) are collected in remote log collection nodes. We collect traces from the block layer of a disk of the DataNodes in such a stage where the applications have submitted block I/O structures to the block device using the *blktrace* [Brunelle (2007)] linux utility. The traces include details such as process id (**pid**), CPU core submitting I/O, logical block address (LBA), size (no. of 512 byte disk blocks), data direction (read/write) information for each I/O request[4]. In the next section, we discuss briefly an example of lineage based application, which we use for our experiments followed by the performance evaluation of LDM.

Lineage Based Application: An example

We use chained MapReduce inter-related jobs to emulate applications which exhibit lineage. Figure 5.9 shows a Lineage application using three MapReduce benchmarks (TeraGen, TeraSort and TeraValidate) which run along with long running non-lineage concurrent applications to form a shared Data Center workload. TeraGen, TeraSort and TeraValidate form logical data-dependent steps to produce the final result.

TeraGen is a data generating job, which produces 1 TB of random data. TeraGen consists of only Map tasks with no reduces. We use TeraGen as a phase which generates data and persTieists it in HDFS, which in-turn is consumed by the next phase, i.e. TeraSort. TeraSort consumes the data generated by TeraGen to sort the data. TeraSort consists of Map phase to read the data, there is one map task per HDFS block. The intermediate data is shuffled and partitioned according to the number of reducers, which are sorted and persisted to HDFS in the reduce phase. The result of TeraSort is used by TeraValidate to validate the output of TeraSort. There are other applications which run concurrently. Consider Table 5.1, we have divided the jobs and applications

[4]Please note, we collected (stored) the traces remotely on a different machine through the network and not stored in the same local HDFS disk for maintaining the purity of the traces & minimize the effects of the SCSI bus [Brunelle (2007); Riska et al. (2007); Chen et al. (2011)].

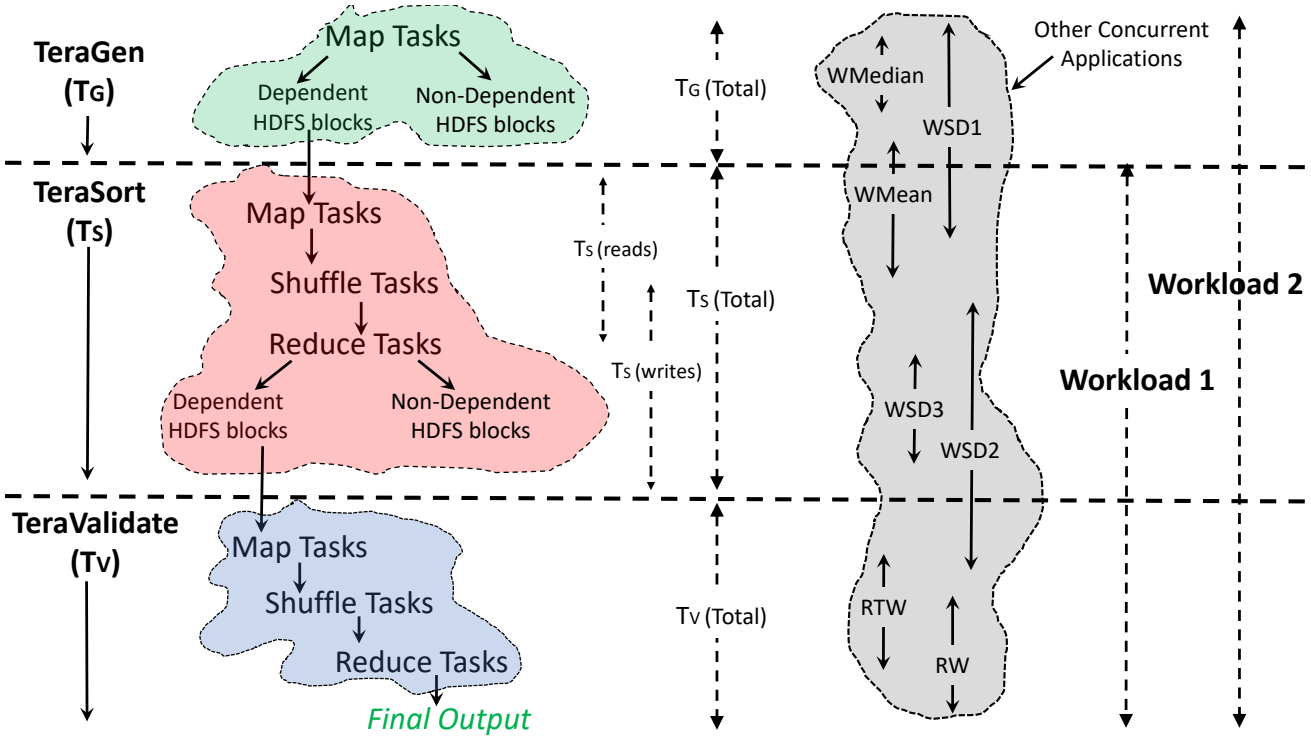


Figure 5.9: Data Center Workload Emulation.

running during TeraGen, TeraSort and TeraValidate into Phase 1, 2, and 3 respectively. We execute combination of these phases to form two types of workloads which exhibit lineage, the details of which is described as follows.

Workload 1: For Workload 1, the lineage application LDM_{W1} is a scenario where the data is already residing in HDFS and stored in HDDs, i.e. we assume data generating phase, Phase 1 *Does Not Exists*. The initial data is retrieved from HDDs and **TeraSort** (T_S), sorts the data in Phase 2 and further the output is required by **TeraValidate** (T_V) in Phase 3 for producing the final result. Therefore, the lineage application LDM_{W1} forms a chain **TeraSort** (T_S) $\rightarrow$ **TeraValidate** (T_V). These applications are batch processing where large chunks of data is already present in storage and at run-time it needs to be acquired. There are a many applications, where data is generated or received on the fly (eg: stream applications, joins, etc) and used subsequently for further processing, which are discussed in Workload 2.

Table 5.1: Experimental Data Center Workloads.

Lineage Application for
Workload 1 "LDM_{W1}": (**TeraSort** (T_S) → **TeraValidate** (T_V)).
Workload 2 "LDM_{W2}": (**TeraGen** (T_G) → **TeraSort** (T_S) → **TeraValidate** (T_V)).

Phase	**Workload 1**	**Workload 2**
Phase 1	*Does Not Exist*	**TeraGen**(T_G), WordStandardDeviation(WSD_1), WordMedian($WMedian$), WordMean($WMean$).
Phase 2	WordStandardDeviation(WSD_1), **TeraSort**(T_S), WordStandardDeviation(WSD_2), WordMean($WMean$), WordStandardDeviation(WSD_3).	WordStandardDeviation(WSD_1), **TeraSort**(T_S), WordStandardDeviation(WSD_2), WordMean($WMean$), WordStandardDeviation(WSD_3).
Phase 3	WordStandardDeviation(WSD_2), **TeraValidate**(T_V), RandomTextWriter(RTW), RandomWriter(RW).	WordStandardDeviation(WSD_2), **TeraValidate**(T_V), RandomTextWriter(RTW), RandomWriter(RW).

Workload 2: For Workload 2, the lineage application LDM_{W2} are such examples of chained jobs, where data is generated on the fly and then subsequent results are produced and consumed. Therefore, the lineage application LDM_{W2} forms a chain **TeraGen** (T_G) → **TeraSort** (T_S) → **TeraValidate** (T_V).

As can be seen in Figure 5.9, for every phase we have marked the Dependent and Non-Dependent HDFS blocks. Dependent HDFS blocks are those blocks which are consumed by the set of tasks in future. While Non-Dependent HDFS blocks are those blocks which are either never consumed or those replicas of Dependent blocks which provide fault-tolerance. The discovery of data dependence and mitigating the impact of this dependency is critical for application performance. With trace-driven experiments and performance evaluation of results in the next section, for both data-center workload scenarios, i.e. Workload 1 and Workload 2, we compare the data-dependency management capability of LDM with the latest implementation of HDFS.

5.3.2 Performance Evaluation

We compare the effectiveness of our dependency mitigation technique scheme, LDM, with the latest implementation of HDFS used deployments for both data-center workload scenarios.

For our experiments, we use the default parameters, which is based on the storage devices and driver specifications. Based on trace-driven simulations, in the next section we analyze the performance of both the schemes, i.e. HDFS and LDM for Data Center workloads 1 and 2.

Total Workload Completion Time

Figure 5.10 represents the total time taken (y-axis) for finishing the data-center workloads. This represents the time to complete all applications in the workload, i.e. lineage as well as other non-lineage applications. It is observed that LDM reduces the time taken to finish the workloads significantly, thereby bridging the deficiencies of HDFS to manage data dependencies. The performance gain (in terms of completion time) is 29% and 52% for workloads 1 and 2, respectively. In LDM, the dependent blocks are identified and placed in SSDs. This serves multi-fold. First, the time to write data and the subsequent future read access is improved as now the dependent blocks do not have to undergo contentions at the disk interface. Secondly, the pipelining of replicas somehow constricts the performance during writes, as the replicas of dependent blocks are placed in HDDs, but placing one (or more) replica in SSD improves the future read time significantly.

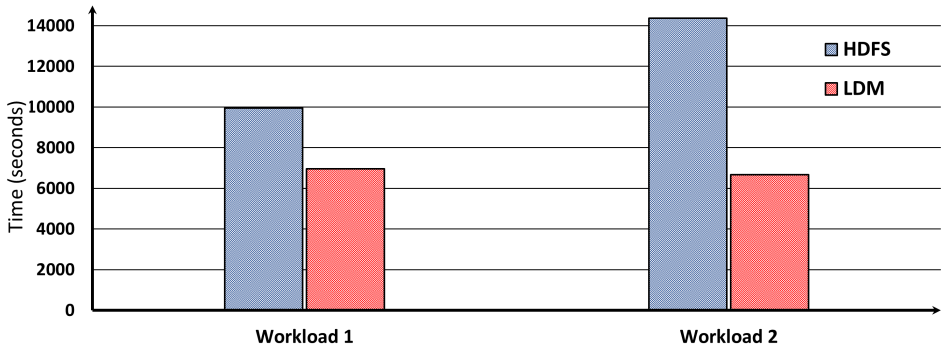


Figure 5.10: Total Time taken by HDFS and LDM (Workload 1 and 2).

It is also observed that though Workload 2 is a subset of Workload 1, and there is more opportunities of optimization via LDM in workload 2, but the gains derived are much more. We discuss in detail with fine grained analysis of each job for both the workloads to understand how LDM optimizes and manages data-dependency which leads to savings in I/O time.

Time taken by chained applications

Figure 5.11 represents the time taken (y-axis) for completing (TeraGen T_G + TeraSort T_S + TeraValidate T_V) and (TeraSort T_S + TeraValidate T_V) by Hadoop (HDFS), and LDM optimizations during workloads 1 and 2, i.e. LDM_{w1} and LDM_{w2}, respectively. It is observed that LDM outperforms HDFS by 55%. The graphs for LDM_{w1} is same, as the data already resides in HDD for consumption for TeraSort, i.e. TeraGen does not exists. Moreover, we also observe that the time taken to complete (TeraSort T_S + TeraValidate T_V) for LDM_{w1} is lower than LDM_{w2} by 40%, though in both cases, we calculate the times from the start of TeraSort and to the finish of TeraValidate.

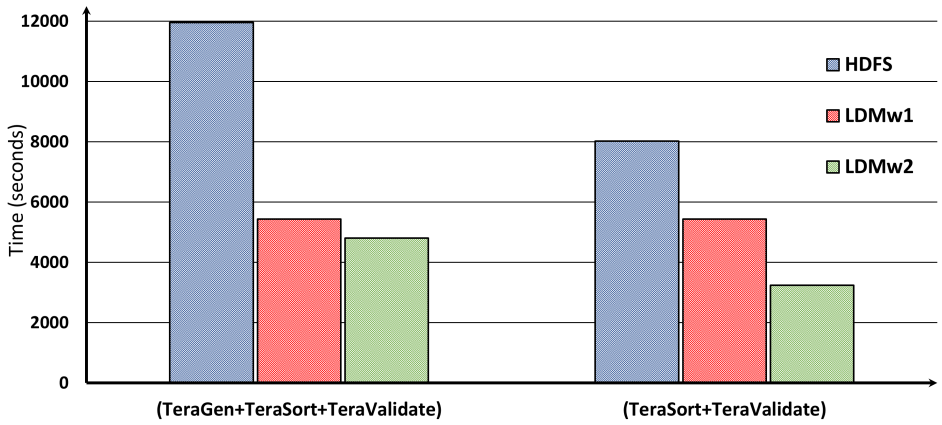


Figure 5.11: Time taken by Chained Applications.

In-order to understand the difference between performance for applications LDM_{w1} and LDM_{w2} from the beginning of TeraSort T_S and end of TeraValidate T_V using LDM, we plot the individual job completion times of TeraGen, TeraSort and TeraValidate, as shown in Figure 5.12.

From Figure 5.12, we observe the following: 1) The time taken by LDM is significantly lower than for all jobs. 2) TeraGen does not exist for LDM_{w1}, as we assume that the data is already residing

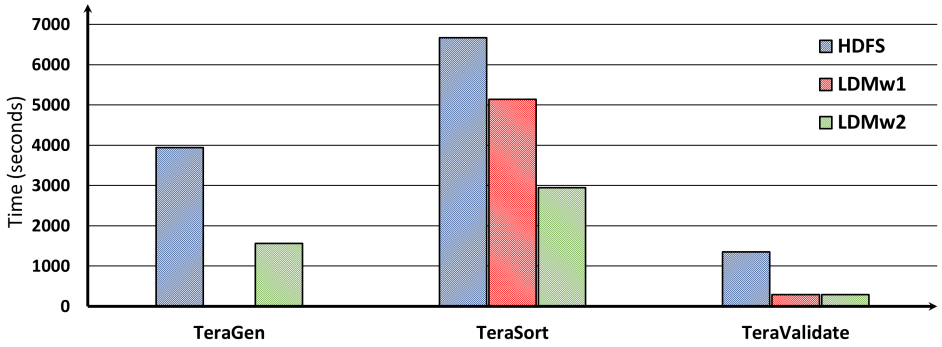


Figure 5.12: Anatomy of Job completion time of Chained MapReduce Applications.

in HDFS. Hence for the analysis of results between LDM optimizations for lineage applications LDM_{w1} and LDM_{w2}, we do not consider the time taken to complete TeraGen. 3) Time taken by TeraValidate for both LDM_{w1} and LDM_{w2} is same, suggesting that most of the difference occurs during TeraSort phase. Therefore, we further investigate the TeraSort job.

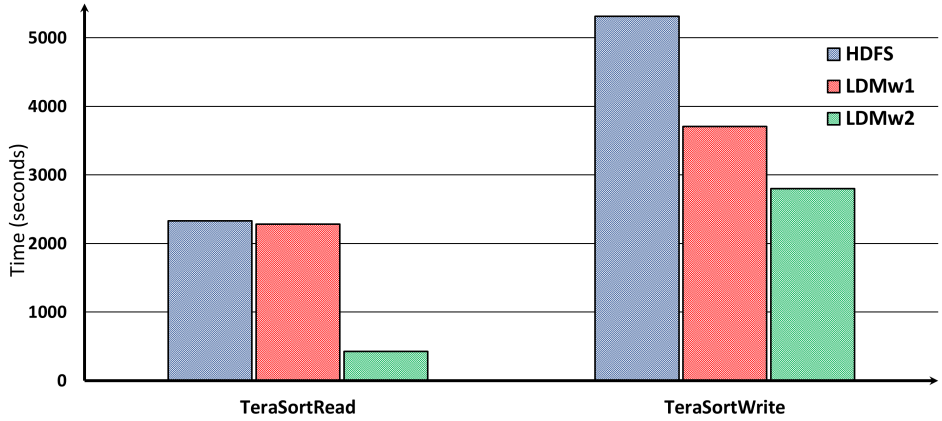


Figure 5.13: Total Read and Write Time of TeraSort job to show the difference between LDM for lineage applications LDM_{w1} and LDM_{w2}.

Figure 5.13 shows the total time taken to read or write, right from the first block of read to the last block (similarly for writes) belonging to TeraSort job, i.e. $(T_s(reads))$ and $(T_s(writes))$ respectively.

For reads, there is marginal gain between HDFS and LDM_{w1}, while there is significant improvements for LDM_{w2}. This is attributed to the characteristic of the job, as most of the reads

occur in the Map phase, for LDM_{w1}, as the data is fetched from HDDs which is nearly same as with no optimization, i.e. HDFS. While for LDM_{w2}, the TeraGen phase is optimized and LDM places (writes) dependent blocks in SSDs, therefore, leading to large reduction in time for reads during TeraSort. We classify the data chunks (128 MB) into dependent and non-dependent, i.e. those HDFS blocks decided by LDM to be placed in SSD and HDD, respectively.

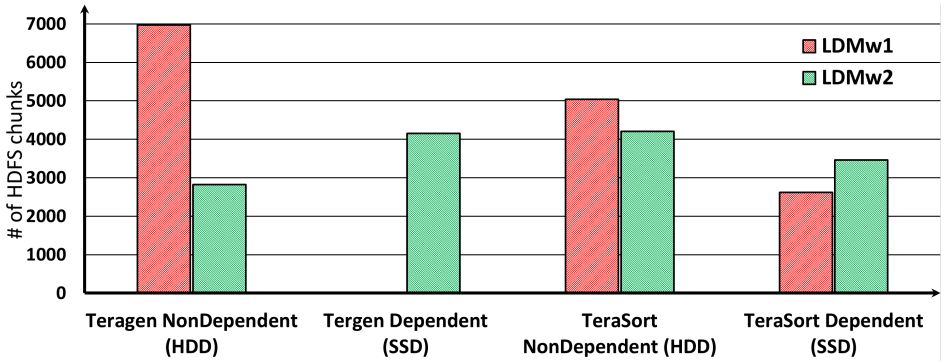


Figure 5.14: Dependent and Non-Dependent blocks.

Figure 5.14 shows the number of HDFS chunks (y-axis) versus the dependent or non-dependent blocks decided by LDM to be placed during that phase. The dependent blocks are written during that phase and consumed in the next. While non-dependent are those data chunks which are those blocks which are not required for computation in near future. We observe that for TeraGen, all blocks are classified non-dependent for LDM_{w1}, as those data sets are already residing in HDD, while none are dependent. For LDM_{w2} application, the dependent blocks are placed in SSD, which are used during the TeraSort run. This justifies the higher read performance for LDM_{w2} than LDM_{w1} for the same LDM optimization observed at the same start and end point (in terms of job completion).

During the writes of TeraSort (refer to Figure 5.13), in both scenarios the time is reduced significantly. It would be expected that the write time for LDM_{w1} and LDM_{w2} should be same, as the difference is only for the reads from SSD (in case of LDM_{w2}) and writes should be the same but LDM_{w2} performs 17% better. This is because of the nature of the dataflow framework

(MapReduce), which does a lot of inter-mediate local writes, which in the case of LDM_{w2} is in SSD, which is faster.

Therefore, we observe that LDM is able to manage the data-dependency while reducing the time taken by over-all workload and chained applications. We further study the impact of LDM on other concurrent non-lineage data center applications

Impact of LDM on other concurrent applications

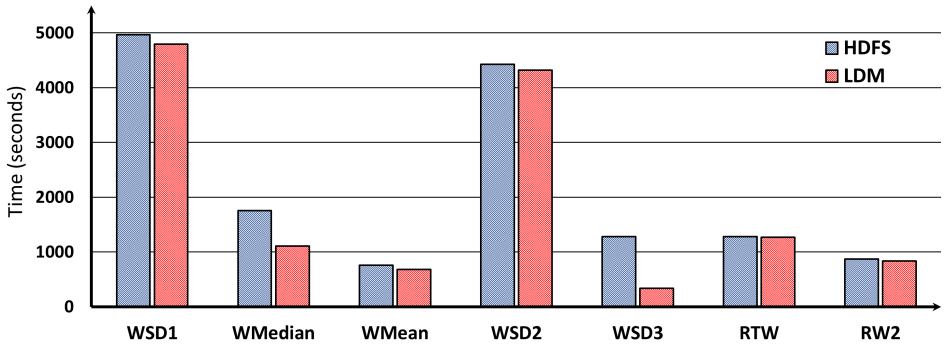


Figure 5.15: Impact of LDM on other concurrent applications running at the same time.

Figure 5.15 represents the time taken by different concurrent non-lineage applications by Hadoop (HDFS) and LDM. It clearly shows that by employing LDM, the I/O time for other concurrent applications also reduces. It forms a *Domino effect*, in which optimizing on type of application also effects the performance of others. This is so because some of the blocks (dependent) which belong to Chained applications are offloaded from the HDD request queue and placed in a different media (SSD). Therefore, there are fewer interferences and the multiplexing of I/O effect is reduced, which leads to over-all savings in execution time for non-lineage concurrent applications.

LDM is designed and developed to mitigate the impact of delays associated to dependency of data in lineage class of applications. Through trace-driven based experiments, LDM shows to successfully orchestrate the application needs apriori and match storage capabilities to deliver performance. There is also evidence that by deploying LDM, the execution time of other applications also reduces. We conclude the chapter in the next section with discussions on future work.

5.4 Conclusion

LDM provides a uniform execution environment across storage and compute, which addresses specific needs of applications with data-dependency (or *lineage*). LDM is a lineage-aware data management system which has an oracle-like deterministic capability to know the future usage of data based on knowledge already present in the data processing framework and ecosystem. These informations and statistics are being produced and consumed by different components of the system. LDM amalgamates these informations from the entire data center ecosystem to dictate tier-aware storage policies for lineage class of applications to mitigate the impact of data dependency for lineage class of applications. Through the development of **block graphs**, LDM is able to capture the complete time-based data-task associations and use it to perform life-cycle management through tiering of data blocks belonging to applications exhibiting lineage. With trace-driven experiments, LDM is able to achieve 29% to 52% reduction in over-all data center workload execution time. Moreover, by deploying LDM with extensive pre-processing creates a efficient data consumption pipelines also shows to reduce the write time and read delays significantly.

In future, we plan to investigate further with LDM for PCIe based NVMe SSDs to leverage parallelism provided by them. We also plan to implement LDM for hybrid set-ups comprising of DAS, NAS and SAN setups with a wide variety of HDDs, and SCMs (with different combinations) and study the data movement impact across networks. LDM opens an avenue for a large diaspora of application and data processing frameworks and we have implemented it for MapReduce environments. It would be interesting to develop LDMs capability to understand various other frameworks like Spark, Parquet, etc. and work together in a unified environment to cater to different syntax and semantics of applications. LDM is effectively able to capture the inherent lineage and utilizes all tiers of storage to reduce data access delays in conjunction with workload-aware tiering by orchestrating multiple data management features. Broader impact of LDM is that it would aid Data Centers to effectively utilize multiple tiers of storage while keeping the Total Cost of Ownership (TCO) low as well as ensuring lower memory and resource footprint leading to energy savings.

CHAPTER 6. CONCLUSION AND FUTURE WORK

In this chapter, we summarize our work on Host Managed Storage Solutions for Big Data with discussions of the impact of this research and discussions on future work. Our contributions are discussed briefly in Section 1.3 (also refer to Figure 1.3).

Our effort has been to deliver near-ideal performance of storage systems, by identifying issues, designing, and, developing software defined storage capabilities with minimal or no infrastructural change for Data Centers experiencing Big Data. Thereby, making changes feasible. Our solutions do not change application characteristics nor improve storage devices or network infrastructures, but only the way data is managed. Therefore, this research aims to improve the layers along the odyssey of data access environment by understanding the I/O hierarchy and the application needs from storage. We have contributed in the following.

1) **Operating System optimizations**, dealing with optimizing the OS and extending its competency. We develop solutions from the core of the operating system, BID-HDD, i.e. a block I/O scheduling scheme to avoid contentions and improve individual storage device capabilities.;

2) **Multi-tier solutions**, which focuses on systems design to incorporate heterogeneous tiers of storage together coupled with value propositions of data being scattered over multiple devices. We manage multiple devices and develop methodologies, BID-Hybrid, to automated tiering using the information obtained at the block interface using SSDs for improving disk performance.;

3) **Workload specific optimizations**, are full-stack data center storage solutions designed and developed to suit workload characteristics. We design and develop methods, LDM- our data management solution, for the complete data center ecosystem using multiple tiers of storage for mitigating the impact of data-dependency in lineage class of applications. LDM amalgamates the information from all the stratas, devices and layers of I/O path.

With theoretical and experimental evaluations, our host managed storage solutions, namely, BID-HDD, BID-Hybrid, and LDM, fulfils our objective of narrowing the gap between what storage is capable of delivering and what it actually delivers in a Big Data environment.

This chapter is organized based on our contributions and their conclusions which form the three sections, namely Section 6.1 for BID-HDD, Section 6.2 for BID-Hybrid and Section 6.3 for LDM.

6.1 Operating System Block Layer Optimizations: BID-HDD

We have developed and designed a contention avoidance scheme for disk based storage devices known as "BID-HDD: Bulk I/O Dispatch" in the Linux block layer, specifically to suit multi-tenant, multi-tasking and skewed shared Big Data deployments. Through trace-driven experiments using in-house developed system simulators and cloud emulating Big Data benchmarks, we show the effectiveness of both our schemes. *BID-HDD*, which is essentially a block I/O scheduling scheme, results in 28% to 52% lesser time for all I/O requests than the best performing Linux disk schedulers, which is discussed in detail in Chapter 3.

From the operating systems perspective, we believe that the I/O kernel data structures (refer to Appendix A) needs to be redesigned as the application needs and storage characteristics have changed dramatically. It would be interesting to develop methodologies for determining optimal kernel I/O data-structure ("BIO") size by analyzing the requirements of combination of workloads. We believe that this along-with efficient I/O packing techniques to have fewer but large requests for such I/O intensive workloads would have significant impact on performance. Additionally, its equally important to study the impact of this change on various layers of the I/O and network infrastructure. This is analogous to the development of TLB-HugePages [Ryoo et al. (2017)].

6.2 Multi-tier Operating System optimizations: BID-Hybrid

We have developed and designed two novel Contention Avoidance storage solutions, collectively known as "BID: Bulk I/O Dispatch" in the Linux block layer, specifically to suit multi-tenant, multi-tasking and skewed shared Big Data deployments. Through trace-driven experiments using

in-house developed system simulators and cloud emulating Big Data benchmarks, we show the effectiveness of both our schemes. In Chapter 3, we have discussed *BID-HDD*, which results in 28% to 52% lesser time for all I/O requests than the best performing Linux disk schedulers. In Chapter 4, we discuss *BID-Hybrid*, tries exploit SSDs superior random performance to further reduce contentions at disk based storage. BID-Hybrid is experimentally shown to be successful in achieving 6% to 23% performance gains over BID-HDD and 33% to 54% over best performing Linux scheduling schemes.

In future, it would be interesting to design a system with BID schemes for block level contention management coupled with self-optimizing block re-organization of BORG Bhadkamkar et al. (2009), adaptive data migration policies of ADLAM [Zhang et al. (2010)], and replication-management of such as Triple-H [Islam et al. (2015)]. This could solve the issue of workload & cost-aware tiering for large scale data-centers experiencing Big Data workloads.

Apart from performance improvements of storage systems, the over-all deployment of BID schemes in data centers would also lead to energy footprint reduction and increase in lifespan expectancy of disk based storage devices.

6.3 Data Management for Lineage based Applications: LDM

In Chapter 5, we discuss LDM, our lineage-aware data management scheme which provides a uniform execution environment across storage and compute, and addresses specific needs of applications with data-dependency (or *lineage*). LDM is a full-stack lineage-aware data management system which has an oracle-like deterministic capability to know the future usage of data based on knowledge already present in the data processing framework and ecosystem. These informations and statistics are being produced and consumed by different components of the system. LDM amalgamates these informations from the entire data center ecosystem to dictate tier-aware storage policies for lineage class of applications to mitigate the impact of data dependency for lineage class of applications. Through the development of **block graphs**, LDM is able to capture the complete time-based data-task associations and use it to perform life-cycle management through tiering of

data blocks belonging to applications exhibiting lineage. With trace-driven experiments, LDM is able to achieve 29% to 52% reduction in over-all data center workload execution time. Moreover, by deploying LDM with extensive pre-processing creates a efficient data consumption pipelines also shows to reduce the write time and read delays significantly.

In future, we plan to investigate further with LDM for PCIe based NVMe SSDs to leverage parallelism offered by them. We also plan to implement LDM for hybrid storage network set-ups comprising of DAS, NAS and SAN with a wide variety of HDDs, and SCMs (with different combinations) and study the data movement impact across networks. LDM opens an avenue for a large diaspora of application and data processing frameworks and we have implemented it for MapReduce environments. It would be interesting to develop LDMs capability in understanding various other frameworks like Spark, Parquet, etc. for working together in a unified environment to cater to different syntax and semantics of applications. LDM is effectively able to capture the inherent lineage and utilizes all tiers of storage to reduce data access delays in conjunction with workload-aware tiering by orchestrating multiple data management features. Broader impact of LDM is that it would aid data centers to effectively utilize multiple tiers of storage while keeping the Total Cost of Ownership (TCO) low as well as ensuring lower memory and resource footprint leading to energy savings.

Finally, we would like to further investigate and develop the field of **"Data assisted systems engineering"**. Throughout this research, we have utilized the information gathered by various layers of the I/O hierarchy to develop storage solutions. Therefore, the data generated from the various components of the system assist in making the performance of Big Data storage systems faster.

BIBLIOGRAPHY

Abad, C. L., Roberts, N., Lu, Y., and Campbell, R. H. (2012). A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 100–109. IEEE.

Afrati, F. N. and Ullman, J. D. (2010). Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM.

Aghayev, A., Tso, T., Gibson, G., and Desnoyers, P. (2017). Evolving ext4 for shingled disks.

Agrawal, S., Narasayya, V., and Yang, B. (2004). Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM.

Alagiannis, I., Idreos, S., and Ailamaki, A. (2014). H2o: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1103–1114. ACM.

Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E. (2011). Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM.

Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I. (2012). Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association.

Arpaci-Dusseau, R. H. and Arpaci-Dusseau, A. C. (2014). *Operating systems: Three easy pieces*, volume 151.

Avanzini, A. (2014). Debugging fanatic, linux and xen enthusiast. bfq i/o scheduler.

Axboe, J. (2004). Linux block iopresent and future. In *Ottawa Linux Symp*, pages 51–61.

Balasubramonian, R., Chang, J., Manning, T., Moreno, J. H., Murphy, R., Nair, R., and Swanson, S. (2014). Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42.

Bhadkamkar, M., Guerra, J., Useche, L., Burnett, S., Liptak, J., Rangaswami, R., and Hristidis, V. (2009). Borg: Block-reorganization for self-optimizing storage systems. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 183–196, Berkeley, CA, USA. USENIX Association.

Bian, H., Yan, Y., Tao, W., Chen, L. J., Chen, Y., Du, X., and Moscibroda, T. (2017). Wide table layout optimization based on column ordering and duplication. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 299–314. ACM.

Bisson, T. and Brandt, S. A. (2007). Reducing hybrid disk write latency with flash-backed i/o requests. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS'07. 15th International Symposium on*, pages 402–409. IEEE.

Bjørling, M., Axboe, J., Nellans, D., and Bonnet, P. (2013). Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA. ACM.

Borba, E. and Tavares, E. (2017). Stochastic modeling for performance and availability evaluation of hybrid storage systems. *Journal of Systems and Software*, 134:1–11.

Brunelle, A. D. (2007). blktrace user guide.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010). Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296.

Chang, H.-P., Liao, S.-Y., Chang, D.-W., and Chen, G.-W. (2015). Profit data caching and hybrid disk-aware completely fair queuing scheduling algorithms for hybrid disks. *Software: Practice and Experience*, 45(9):1229–1249.

Chen, F., Koufaty, D. A., and Zhang, X. (2011). Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing*, pages 22–32, New York, NY, USA. ACM.

Choi, D., Jeon, M., Kim, N., and Lee, B.-D. (2017). An enhanced data-locality-aware task scheduling algorithm for hadoop applications. *IEEE Systems Journal*.

Chu, W. W. (1969). Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers*, 100(10):885–889.

Cornell, D. W. and Yu, P. S. (1990). An effective approach to vertical partitioning for physical design of relational databases. *IEEE Transactions on Software engineering*, 16(2):248–258.

Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113.

Di Mauro, M., Longo, M., Postiglione, F., Carullo, G., and Tambasco, M. (2017). Software defined storage: Availability modeling and sensitivity analysis. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS), 2017 International Symposium on*, pages 1–7. IEEE.

Eshghi, K. and Micheloni, R. (2013). Ssd architecture and pci express interface. In *Inside Solid State Drives (SSDs)*, pages 19–45.

Ferdaus, M. H., Murshed, M., Calheiros, R. N., and Buyya, R. (2017). An algorithm for network and data-aware placement of multi-tier applications in cloud data centers. *Journal of Network and Computer Applications*, 98:65–83.

Galaktionov, V., Chernishev, G., Smirnov, K., Novikov, B., and Grigoriev, D. A. (2016). A study of several matrix-clustering vertical partitioning algorithms in a disk-based environment. In *International Conference on Data Analytics and Management in Data Intensive Domains*, pages 163–177. Springer.

Gao, Y., Zhang, H., Tang, B., Zhu, Y., and Ma, H. (2017). Two-stage data distribution for distributed surveillance video processing with hybrid storage architecture. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 616–623. IEEE.

Gkantsidis, C., Vytiniotis, D., Hodson, O., Narayanan, D., Dinu, F., and Rowstron, A. I. (2013). Rhea: Automatic filtering for unstructured cloud storage. In *NSDI*, volume 13, pages 2–5.

Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., and Madden, S. (2010). Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116.

Gunda, P. K., Ravindranath, L., Thekkath, C. A., Yu, Y., and Zhuang, L. (2010). Nectar: Automatic management of data and computation in datacenters. In *OSDI*, volume 10, pages 1–8.

Guo, S., Xiong, J., Wang, W., and Lee, R. (2012). Mastiff: A mapreduce-based system for time-based big data analytics. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 72–80. IEEE.

Harter, T., Borthakur, D., Dong, S., Aiyer, A., Tang, L., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2014). Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, Santa Clara, CA. USENIX.

He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., and Xu, Z. (2011). Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE.

Herodotou, H. (2016). Towards a distributed multi-tier file system for cluster computing. In *Data Engineering Workshops (ICDEW), 2016 IEEE 32nd International Conference on*, pages 131–134. IEEE.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22.

Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. (2010). The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51.

Ibrahim, S., Jin, H., Lu, L., He, B., and Wu, S. (2011). Adaptive disk i/o scheduling for mapreduce in virtualized environment. In *2011 International Conference on Parallel Processing*, pages 335–344.

Iliadis, I., Jelitto, J., Kim, Y., Sarafijanovic, S., and Venkatesan, V. (2015). Exaplan: Queueing-based data placement and provisioning for large tiered storage systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 218–227.

Inc., R. H. (2015). Linux performance tuning guide, red-hat enterprise.

Islam, N. S., Lu, X., ur Rahman, M. W., Shankar, D., and Panda, D. K. (2015). Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 101–110.

Islam, N. S., Wasi-ur Rahman, M., Lu, X., and Panda, D. K. D. (2016). Efficient data access strategies for hadoop and spark on hpc cluster with heterogeneous storage. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 223–232. IEEE.

Iyer, S. and Druschel, P. (2001). Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 117–130, New York, NY, USA. ACM.

Jahani, E., Cafarella, M. J., and Ré, C. (2011). Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396.

Ji, C., Chang, L.-P., Wu, C., Shi, L., and Xue, C. J. (2017). An i/o scheduling strategy for embedded flash storage devices with mapping cache. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Jindal, A. and Dittrich, J. (2011). Relax and let the database do the partitioning online. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 65–80. Springer.

Jindal, A., Palatinus, E., Pavlov, V., and Dittrich, J. (2013). A comparison of knives for bread slicing. *Proceedings of the VLDB Endowment*, 6(6):361–372.

Jindal, A., Quiané-Ruiz, J.-A., and Dittrich, J. (2011). Trojan data layouts: right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 21. ACM.

Joo, Y., Park, S., and Bahn, H. (2017). Exploiting i/o reordering and i/o interleaving to improve application launch performance. *Trans. Storage*, 13(1):8:1–8:17.

Kakoulli, E. and Herodotou, H. (2017). Octopusfs: A distributed file system with tiered storage management. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 65–78. ACM.

Kang, Y., Kee, Y.-s., Miller, E. L., and Park, C. (2013). Enabling cost-effective data processing with smart ssd. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pages 1–12. IEEE.

Khasnabish, J. N., Mithani, M. F., and Rao, S. (2017). Tier-centric resource allocation in multi-tier cloud systems. *IEEE Transactions on Cloud Computing*, 5(3):576–589.

Kim, J., Lee, E., and Noh, S. H. (2016). I/o scheduling schemes for better i/o proportionality on flash-based ssds. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 221–230. IEEE.

Kim, Y., Gupta, A., Urgaonkar, B., Berman, P., and Sivasubramaniam, A. (2011). Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 227–236.

Koller, R. and Rangaswami, R. (2010). I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):13.

Krish, K., Iqbal, M. S., and Butt, A. R. (2014a). Venu: Orchestrating ssds in hadoop storage. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 207–212. IEEE.

Krish, K., Khasymski, A., Butt, A. R., Tiwari, S., and Bhandarkar, M. (2013). Aptstore: Dynamic storage management for hadoop. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 33–41. IEEE.

Krish, K., Wadhwa, B., Iqbal, M. S., Rafique, M. M., and Butt, A. R. (2016). On efficient hierarchical storage for big data processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 403–408. IEEE.

Krish, K. R., Anwar, A., and Butt, A. R. (2014b). hats: A heterogeneity-aware tiered storage for hadoop. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 502–511.

Lee, M., Kang, D. H., Lee, M., and Eom, Y. I. (2017). Improving read performance by isolating multiple queues in nvme ssds. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, page 36. ACM.

Lee, S., Jo, J.-Y., and Kim, Y. (2016). Performance improvement of mapreduce process by promoting deep data locality. In *Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on*, pages 292–301. IEEE.

LeFevre, J., Sankaranarayanan, J., Hacigumus, H., Tatemura, J., Polyzotis, N., and Carey, M. J. (2014). Miso: souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1591–1602. ACM.

Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. (2014). Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM.

Li, Y. and Patel, J. M. (2014). Widetable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment*, 7(10):907–918.

Lin, L., Zhu, Y., Yue, J., Cai, Z., and Segee, B. (2011). Hot random off-loading: A hybrid storage system with dynamic data migration. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 318–325.

Liu, Y., Huang, J., Xie, C., and Cao, Q. (2010). Raf: A random access first cache management to improve ssd-based disk cache. In *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, pages 492–500.

Lu, W., Wang, Y., Jiang, J., Liu, J., Shen, Y., and Wei, B. (2017). Hybrid storage architecture and efficient mapreduce processing for unstructured data. *Parallel Computing*, 69:63–77.

Ma, L. and Xu, L. (2016). Hmss: A high performance host-managed shingled storage system based on awareness of smr on block layer. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on*

Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on, pages 570–577. IEEE.

Malladi, K. T., Awasthi, M., and Zheng, H. (2016). Flexdrive: A framework to explore nvme storage solutions. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, pages 1115–1122. IEEE.

Mandal, S., Kuenning, G., Ok, D., Shastry, V., Shilane, P., Zhen, S., Tarasov, V., and Zadok, E. (2016). Using hints to improve inline block-layer deduplication. In *FAST*, pages 315–322.

March, S. T. and Rho, S. (1995). Allocating data and operations to nodes in distributed database design. *IEEE Transactions on Knowledge and data engineering*, 7(2):305–317.

Mihailescu, M., Soundararajan, G., and Amza, C. (2012). Mixapart: decoupled analytics for shared storage systems. In *In USENIX FAST*.

Mishra, P., Mishra, M., and Somani, A. K. (2016). Bulk i/o storage management for big data applications. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*, pages 412–417. IEEE.

Mishra, P., Mishra, M., and Somani, A. K. (2017). Applications of hadoop ecosystem tools. In *Handbook of NoSQL: Database for Storage and Retrieval of Data in Cloud, ISBN 9781498784368*. Taylor and Francis Group CRC Press.

Mishra, P. and Somani, A. K. (2017). Host managed contention avoidance storage solutions for big data. *Journal of Big Data*, 4(18).

Mittal, S. and Vetter, J. S. (2016). A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550.

Moon, S., Lee, J., Sun, X., and Kee, Y.-S. (2015). Optimizing the hadoop mapreduce framework with high-performance storage devices. *J. Supercomput.*, 71(9):3525–3548.

Nanavati, M., Schwarzkopf, M., Wires, J., and Warfield, A. (2015). Non-volatile storage. *Queue*, 13(9):20:33–20:56.

Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. (1984). Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710.

Olson, J. T., Patil, S. R., Shiraguppi, R. M., and Spear, G. A. (2017). Handling data block migration to efficiently utilize higher performance tiers in a multi-tier storage environment. US Patent App. 15/499,274.

Park, H., Yoo, S., Hong, C.-H., Yoo, C., undefined, undefined, undefined, and undefined (2016). Storage sla guarantee with novel ssd i/o scheduler in virtualized data centers. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2422–2434.

Park, S. and Shen, K. (2012). Fios: a fair, efficient flash i/o scheduler. In *FAST*, page 13.

Pfefferle, J., Stuedi, P., Trivedi, A., Metzler, B., Koltsidas, I., and Gross, T. R. (2015). A hybrid i/o virtualization framework for rdma-capable network interfaces. In *ACM SIGPLAN Notices*, volume 50, pages 17–30. ACM.

Riedel, E., Gibson, G., and Faloutsos, C. (1998). Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer.

Riska, A., Larkby-Lahet, J., and Riedel, E. (2007). Evaluating block-level optimization through the io path. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 19:1–19:14, Berkeley, CA, USA. USENIX Association.

Roussos, K. (2007). Storage virtualization gets smart. *Queue*, 5(6):38–44.

Ruemmler, C. and Wilkes, J. (1994). An introduction to disk drive modeling. *Computer*, 27(3):17–28.

Ryoo, J. H., Gulur, N., Song, S., and John, L. K. (2017). Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 469–480. ACM.

Schnberger, G. (2015). Linux i / o scheduler, thomas krenn.

Seelam, S., Romero, R., Teller, P., and Buros, B. (2005). Enhancements to linux i/o scheduling. In *Proc. of the Linux Symposium*, volume 2, pages 175–192.

Shi, H., Arumugam, R. V., Foh, C. H., and Khaing, K. K. (2012). Optimal disk storage allocation for multi-tier storage system. In *APMRC, 2012 Digest*, pages 1–7. IEEE.

Spivak, A., Razumovskiy, A., Nasonov, D., Boukhanovsky, A., and Redice, A. (2017). Storage tier-aware replicative data reorganization with prioritization for efficient workload processing. *Future Generation Computer Systems*.

Tiwari, D., Vazhkudai, S. S., Kim, Y., Ma, X., Boboila, S., and Desnoyers, P. J. (2012). Reducing Data Movement Costs Using Energy-Efficient, Active Computation on SSD. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*.

TPC(tm) (2016). Tpc express benchmark(tm) hs (tpcx-hs)-hadoop suite overview.

Valente, P. and Andreolini, M. (2012). Improving application responsiveness with the bfq disk i/o scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 6:1–6:12, New York, NY, USA. ACM.

Vangoor, B. K. R., Tarasov, V., and Zadok, E. (2017). To fuse or not to fuse: Performance of user-space file systems. In *Proceedings of FAST17: 15th USENIX Conference on File and Storage Technologies*, page 59.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA. ACM.

Wang, H., Huang, P., He, S., Zhou, K., Li, C., and He, X. (2013). A novel i/o scheduler for ssd with improved performance and lifetime. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–5. IEEE.

White, T. (2012). *Hadoop: The definitive guide*.

Xiao, W., Lei, X., Li, R., Park, N., and Lilja, D. J. (2012). Pass: A hybrid storage system for performance-synchronization tradeoffs using ssds. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 403–410.

Xie, Y., Muniswamy-Reddy, K.-K., Feng, D., Long, D. D. E., Kang, Y., Niu, Z., and Tan, Z. (2011). Design and evaluation of oasis: An active storage framework based on t10 osd standard. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE.

Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., and Balakrishnan, V. (2015). Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM.

Yang, Y. and Zhu, J. (2016a). Write skew and zipf distribution: evidence and implications. *Trans Storage*, 12.

Yang, Y. and Zhu, J. (2016b). Write skew and zipf distribution: Evidence and implications. *Trans. Storage*, 12(4):21:1–21:19.

Ye, F., Chen, J., Fang, X., Li, J., and Feng, D. (2015). A regional popularity-aware cache replacement algorithm to improve the performance and lifetime of ssd-based disk cache. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 45–53.

Yi, M., Lee, M., and Eom, Y. I. (2017). Cffq: I/o scheduler for providing fairness and high performance in ssd devices. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, IMCOM '17, pages 87:1–87:6, New York, NY, USA. ACM.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. *HotCloud*, 10:10–10.

Zhang, G., Chiu, L., and Liu, L. (2010). Adaptive data migration in multi-tiered storage based cloud environment. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 148–155. IEEE.

Zheng, D., Burns, R., and Szalay, A. S. (2013). Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 69. ACM.

Zhou, J., Xie, W., Noble, J., Echo, K., and Chen, Y. (2016). Suora: A scalable and uniform data distribution algorithm for heterogeneous storage systems. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*, pages 1–10. IEEE.

APPENDIX A. BLOCK I/O DATA-STRUCTURES

The generic block layer converts I/O requests to I/O operations known as block I/O (or BIO) structures. The BIO data-structure are contiguous disk blocks and contain information such as type of operation (read/write), Logical Block address (LBA) and linked-list of structures known as bio_vecs (which are pages in memory from which the I/O operation needs to be performed.) The I/O scheduler is responsible for creation and merging of "request" structures from BIO structures as well as management of the "request queue."

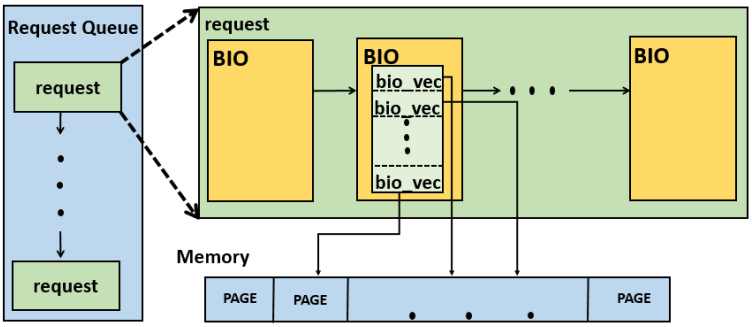


Figure A.1: Request queue processing structures.

Figure A.1 shows the Linux Kernal I/O data structures. The "request queue" is a linked-list of requests structures. Each request structure is a linked-list of BIO (Block I/O) structures, which in-turn are linked-list of bio_vec structures. As per the Linux Kernel 4.15 source code, each bio_vec represents a page in memory, by default it is 4096 bytes. Each BIO structure can contain a maximum of 256 bio_vec structures.

The size of each request structure (max_sectors_kb) depends on the specifications of the hardware, by default it is 512 KB [Inc. (2015)]. The request queue length is limited by the number of read or write request structures present in it. The maximum number of read/write structures in request queue being 128 (nr_requests), while the maximum total structures permitted is 256 (128 reads, 128 writes) [Inc. (2015)].