Journal of Big Data

RESEARCH Open Access

Host managed contention avoidance storage solutions for Big Data

Pratik Mishra* and Arun K. Somani

*Correspondence:
mishrap@iastate.edu
Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

Abstract

The performance gap between compute and storage is fairly considerable. This results in a mismatch between the application needs from storage and what storage can deliver. The full potential of storage devices 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 Big Data shared resource environments. 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 it's significance, the block layer, essentially the block I/O scheduler, hasn't evolved to meet the needs of Big Data. We have designed and developed two contention avoidance storage solutions, collectively known as "BID: Bulk I/O Dispatch" in the Linux block layer specifically to suit multi-tenant, multi-tasking shared Big Data environments. 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. In the first part, we propose a dynamically adaptable block I/O scheduling scheme *BID-HDD* for disk based storage. *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 the effectiveness of *BID-HDD* which results in 28–52% lesser time for all I/O requests than the best performing Linux disk schedulers. In the second part, we propose a hybrid scheme *BID-Hybrid* to exploit SCM's (SSDs) superior random performance to further avoid contentions at disk based storage. *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. This results in performance gain of 6–23% for MapReduce workloads when compared to BID-HDD and 33–54% 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.

Keywords: Multi-tier, Hard disk drives, Solid state drives, MapReduce, Hadoop, Hdfs, Contention avoidance, Big Data, Storage, Block I/O layer, I/O scheduler

Introduction

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 [1]. In cloud environments, with the aid of server and storage virtualization, multiple processes contend for the same physical resource (namely, compute, network and storage) [2]. This causes contentions. In-order to meet their service level agreements (SLAs), cloud providers need to ensure performance isolation gaurantees for every application [3].

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 [4]. The problems associated due to the inefficiencies in data management get amplified in multi-tasking, and shared Big Data environments.

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 1 shows the importance of the block layer. 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.

Unfortunately, despite it's significance, the block layer, essentially the block I/O scheduler hasn't 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"*, specifically to suit multi-tenant, multi-tasking Big Data shared resource environments.

Big Data applications use data processing frameworks such as *Hadoop MapReduce*, which access storage in large data chunks (64 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

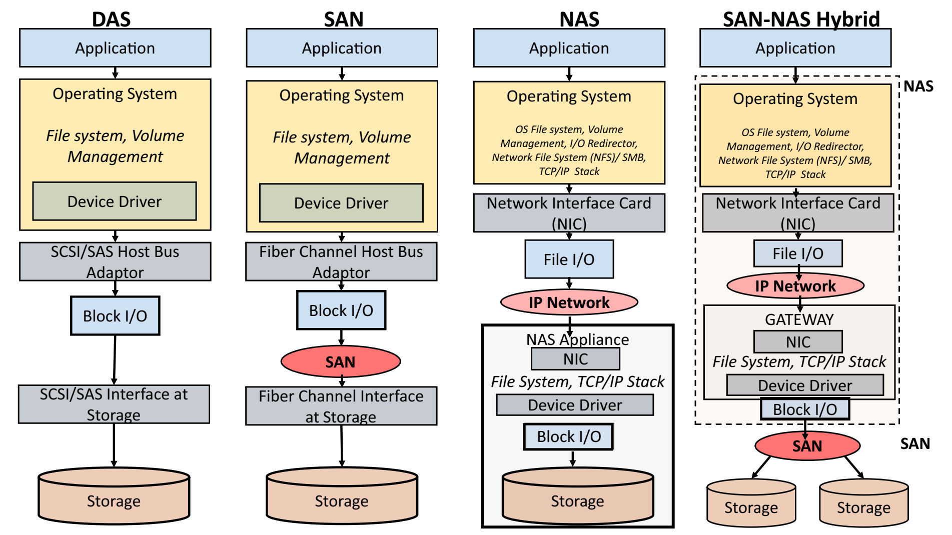

DAS
Application
Operating System
File system, Volume Management
Device Driver
SCSI/SAS Host Bus Adaptor
Block I/O
SCSI/SAS Interface at Storage
Storage
SAN
Application
Operating System
File system, Volume Management
Device Driver
Fiber Channel Host Bus Adaptor
Block I/O
SAN
Fiber Channel Interface at Storage
Storage
NAS
Application
Operating System
OS File system, Volume Management, I/O Redirector, Network File System (NFS)/ SMB, TCP/IP Stack
Network Interface Card (NIC)
File I/O
IP Network
NAS Appliance
NIC
File System, TCP/IP Stack
Device Driver
Block I/O
Storage
SAN-NAS Hybrid
Application
NAS
Operating System
OS File system, Volume Management, I/O Redirector, Network File System (NFS)/ SMB, TCP/IP Stack
Network Interface Card (NIC)
File I/O
IP Network
GATEWAY
NIC
File System, TCP/IP Stack
Device Driver
Block I/O
SAN
SAN
Storage
Storage

Fig. 1 Networked storage architecture

phenomenons such as multiplexing and interleavings, thereby breaking of large data accesses [5–7]. The increase in latency of storage devices (HDDs) adversely affects overall system performance (CPU wait time increase) [8].

In the first solution, we propose a dynamically adaptable Block I/O scheduling scheme *BID-HDD*, for disk based storage. 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 28–52% I/O time performance gain for all I/O requests than the best performing Linux disk schedulers.

With recent developments in NVMe (non-volatile memory) devices such as solid state drives (SSDs), commonly known as storage class memories (SCM) [9], 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 [1, 3]. In the second part, 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-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 6–23% for MapReduce workloads over BID-HDD and 33–54% over the best performing Linux scheduling schemes.

Broader impact of this research would aid Data Centers to achieve their SLAs as well keeping total-cost of ownership (TCO) low. Apart from performance improvements of storage devices, the over-all deployment of BID schemes in data centers would also lead to energy footprint reduction as well as increase in the lifespan expectancy of disk based storage devices.

The rest of the paper is organized as follows. "Background" section provides a brief overview of the working of the I/O stack, secondary memory devices and their inefficiencies in shared large data processing infrastructure. "Requirements from a block I/O scheduler in Big Data deployments" section 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 "Our approach: BID Bulk I/O Dispatch" and "Experiments and performance evaluation" sections, we present our Contention Management schemes followed by our design of experiments and performance evaluation, respectively. "Related works" section provides an in-depth survey of related literature. We conclude the paper in "Conclusion and future works" section with a discussion on future work.

Background

In this section, we first briefly present the working of the Linux I/O stack in "Linux I/O stack" section followed by the additional features of the OS block layer in "OS block layer: additional features" section. In "Secondary storage (block device) characteristics" section , we discuss the physical characteristics of secondary storage devices like HDDs and SCMs used in modern data centers. "Hadoop MapReduce: working and workload characteristics" and "Requirements from a block I/O scheduler in Big Data deployments" sections discuss the I/O workload characteristics of Hadoop deployments and the requirements from a I/O scheduler in such environments, respectively. "Issues with current I/O schedulers" section describes the working of the current state-of-the-art Linux disk schedulers deployed in shared Big Data infrastructure.

Linux I/O stack

The I/O stack of the data center architectures as shown in Fig. 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 [8, 10, 11]. Figure 2 is the simplistic representation of the Linux I/O stack [10, 12].

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:

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 [10, 13, 14]. VFS locates and determines the storage device as well as the file

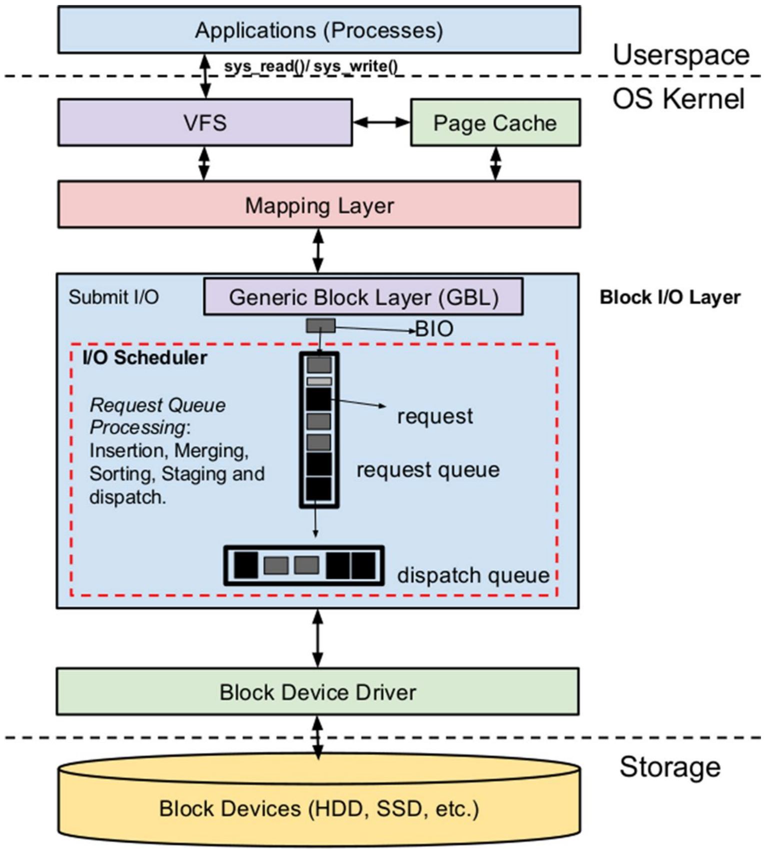


Fig. 2 Architecture of Linux Kernel I/O stack

system hosting the data, starting from a relative location. VFS provides an uniform interface to access multiple file systems [15].

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 ("Conclusion and future works" section) 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. "Issues with current I/O schedulers" section 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 (and size) in which data is accessed from the block device. Therefore, the block layer is the most critical part of the I/O hierarchy.

OS block layer: additional features

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 throughout the paper. Additionally, the block layer has the following benefits which make it suitable for developing SDS solutions:

- *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 [8, 10].

- *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 [8, 10], thereby providing opportunities to exploit information for intelligent optimizations.
- *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 Fig. 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.

Very recently, an important piece of work Bjørling et al. [10] 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 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 SDS solutions for disk based storage devices.

Secondary storage (block device) 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)] [16].

The data in HDDs is organized as 512 byte (or 4 kB 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) [17]. 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.

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 [5, 8, 10].

Due to the physical limitations of HDDs, there have been recent efforts [1, 2, 11, 18–21] in incorporating flash based storage such as SSDs in data centers. The high-speed, non-volatile storage devices like SSDs typically referred to as SCMs access data via electrical signals, as opposed to physical disk arm movement in the case of HDDs [3, 9]. 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.

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 [3]. 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 [12, 22].

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 [1, 9]. 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 [3, 4, 23].
- *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) [9, 19, 24]. 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 [9]. The skew in performance makes it more economically feasible to use HDDs.

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. SCMs are used as cache for disk based storage, coupled with workload-aware tiering [1, 3, 4, 25] for automatic classification of data to balance cost, performance and capacity.

Hadoop MapReduce: working and workload characteristics

Hadoop MapReduce [26, 27] 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 [11, 28]. 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 [27].

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 [23]. Thus, the effects of file system caching is negligible in HDFS [23, 29]. 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 requirements [28]. 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 [5, 6, 28, 29]. This also results in higher CPU wait/idle time as it has to wait for the data [3, 5, 8, 10]. In order to provide performance isolation to each process as well as improve system performance, it is imperative to remove or avoid contentions.

"Issues with current I/O schedulers" section 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.

Requirements from a block I/O scheduler 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 [27] (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 [2]. Additionally, each MapReduce task itself has varying and interleaving I/O characteristics in its Map, Reduce and Shuffle phases [28]. 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 SLAs, it is highly imperative to provide I/O performance isolation for each application [2, 7]. 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.

Issues with current I/O schedulers

Since version 2.6.33, Linux [2, 6, 28] currently employs three disk I/O Schedulers namely Noop, Deadline and Completely Fair Queuing CFQ.

As observed in "Linux I/O stack" section, 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 ("Conclusion and future works" section) 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 HDD with geometry of 1 platter, 100 sectors/track and 100 tracks/platter (see Fig. 3). Consider three processes with process id's (*pid*) *A, B, C* submitting I/O requests to the disk block layer in the order shown in Table 1 (from top to bottom).

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:

Noop Noop is the simplest of the three scheduling algorithms. Figure 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.

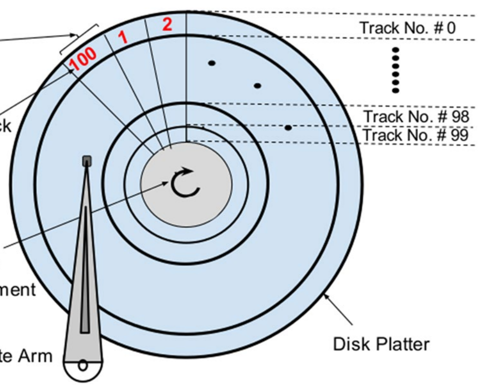


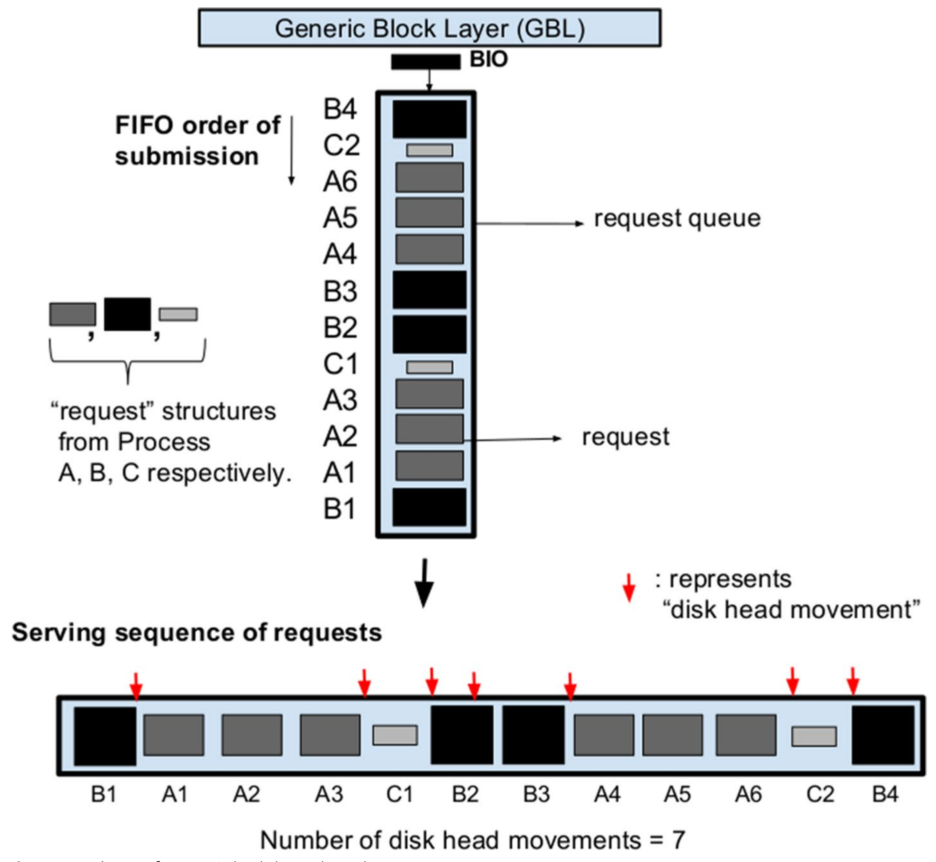
Fig. 3 Geometry of the HDD with 1 platter, 100 sectors/track and 100 tracks/platter

Table 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

A_n, B_n, C_n *n*th request of processes A, B, C submitted to the "request queue", *LBA* starting logical block address (LBA) 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

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.


Generic Block Layer (GBL)
BIO
FIFO order of submission
B4
C2
A6
A5
A4
B3
B2
C1
A3
A2
A1
B1
request queue
request
"request" structures from Process A, B, C respectively.
: represents "disk head movement"
Serving sequence of requests
B1 A1 A2 A3 C1 B2 B3 A4 A5 A6 C2 B4
Number of disk head movements = 7
Fig. 4 Working of *Noop Scheduling Algorithm*

Deadline Scheduler The Deadline Scheduler tries to prevent starvation of requests. Each request is assigned an expiration time (reads = 500 ms, writes = 5000 ms) [6, 28]. 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 [2, 30]. Some implementation have just three queues: 2 FIFO queues and a common Sorted Queue [6]. For simplicity, we consider the former implementation with both FIFO and Sorted queues having separate Read and Write queues as shown in Fig. 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 [6, 13, 31].

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 Fig. 5.

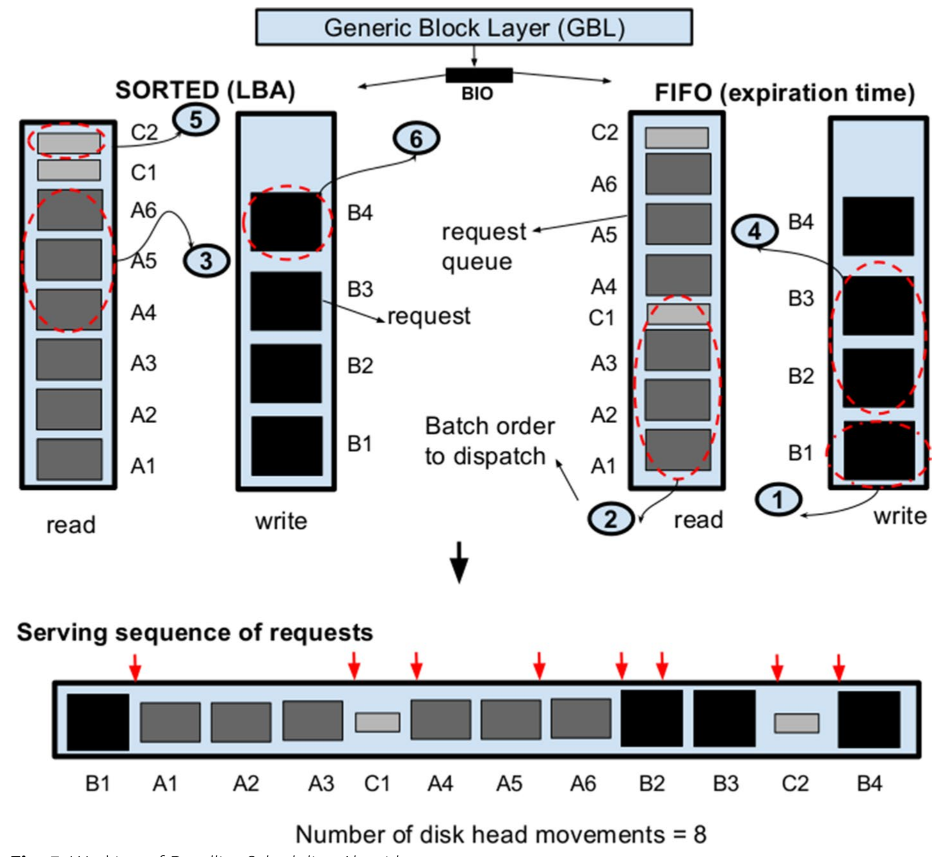

Generic Block Layer (GBL)
BIO
SORTED (LBA)
FIFO (expiration time)
C2
C1
A6
A5
A4
A3
A2
A1
B4
B3
B2
B1
request queue
request
Batch order to dispatch
read
write
1
2
3
4
5
6
Serving sequence of requests
B1 A1 A2 A3 C1 A4 A5 A6 B2 B3 C2 B4
Number of disk head movements = 8

Fig. 5 Working of *Deadline Scheduling Algorithm*

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 and 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 [32]. 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 [2, 6] for highly sequential and multi-process workloads like Hadoop MapReduce.

Completely fair queuing (CFQ) CFQ is the default disk I/O scheduler in the current Linux distribution [2, 7]. It divides the available I/O bandwidth among all the contending I/O request submitting processes [6]. 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* [31]. CFQ is suitable for environments where all processes need equal and periodic share of the block device like interactive applications.

In Fig. 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 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 Fig. 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

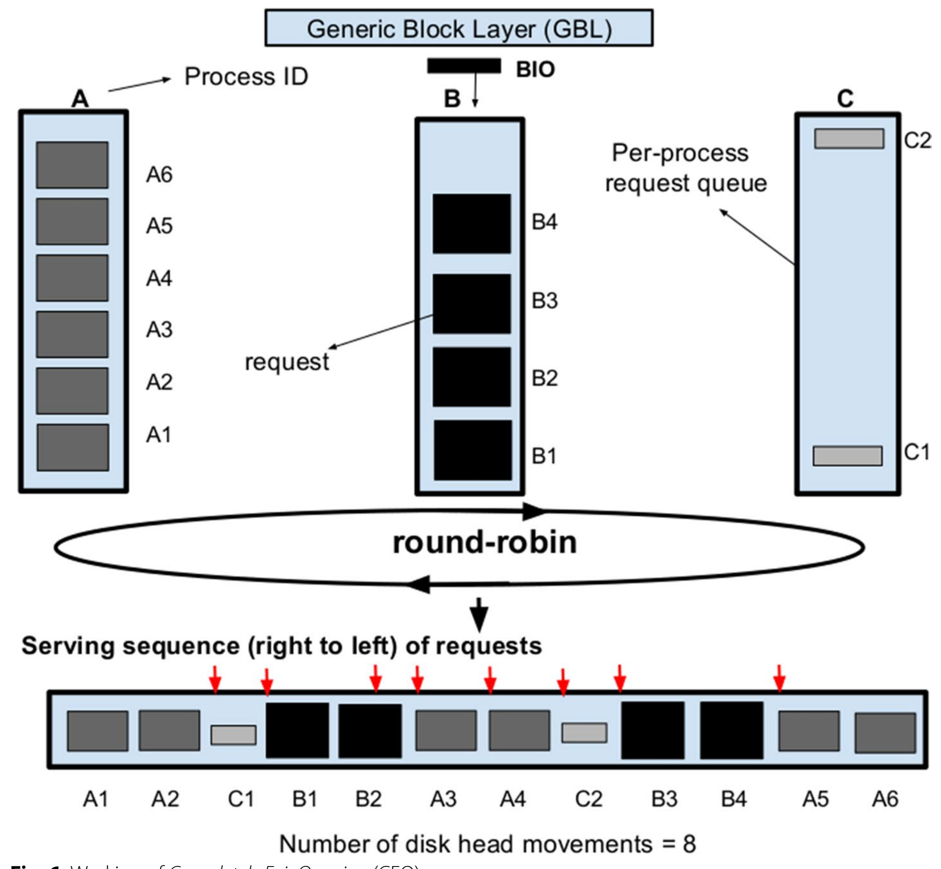

Generic Block Layer (GBL)
BIO
Process ID
A
B
C
A6
A5
A4
A3
A2
A1
B4
B3
B2
B1
C2
C1
request
Per-process request queue
round-robin
Serving sequence (right to left) of requests
A1 A2 C1 B1 B2 A3 A4 C2 B3 B4 A5 A6
Number of disk head movements = 8

Fig. 6 Working of *Completely Fair Queuing* (CFQ)

is undesirable for a multi-process environment with diverse disk I/O characteristics within request queue contending processes [2, 6, 7].

A fourth I/O Scheduling scheme, Anticipatory Scheduler has been discontinued from the Linux kernel. It associates a fixed waiting time (6 ms) for every synchronous (read) request [2, 6, 28]. 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 [5] 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 "Requirements from a block I/O scheduler in Big Data deployments" section and there is a clear need of new I/O scheduling scheme for such Big Data deployments.

Figure 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.

Our approach: BID "Bulk I/O Dispatch"

HDDs form the backbone of data centers storage. The effects of caching is negligible in an enterprise Big Data environment [23, 29] (refer to "Hadoop MapReduce: working and workload characteristics" section), 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 two Contention Avoidance Storage solutions, collectively known as "BID: Bulk I/O Dispatch" in the Linux block layer specifically to suit multi-tenant, multi-tasking shared Big Data environments. In the first part of this section, we propose a Block I/O scheduling scheme *BID-HDD* for disk based storage. BID-HDD tries to recreate the sequentiality in I/O access in order to provide performance isolation to each I/O submitting process.

In the second part, 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 done 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 [10] FIFO architecture for SSD I/O request processing.

BID schemes are designed taking into consideration the requirements laid out earlier in "Requirements from a block I/O scheduler in Big Data deployments" section. BID as a whole is aimed to avoid contentions for storage I/Os following system constraints without compromising the SLAs.

BID-HDD: contention avoiding I/O scheduling for HDDs

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

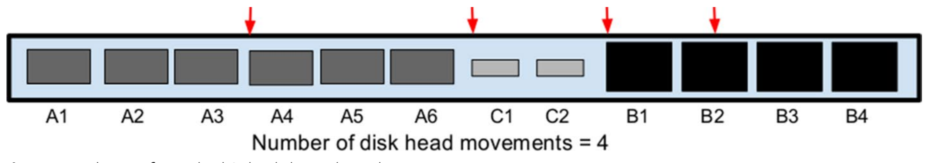


Fig. 7 Working of *an Ideal Scheduling Algorithm*

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.

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 oppurtunuties 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 LBAs. The requests from the *dispatch queue* are dequeued according to the disk controller on the physical device.

Figure 8 shows the working of BID-HDD with the help of the I/O submission order as in Table 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 inserts the block I/O request structures to the *request queue* [10]. 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.

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

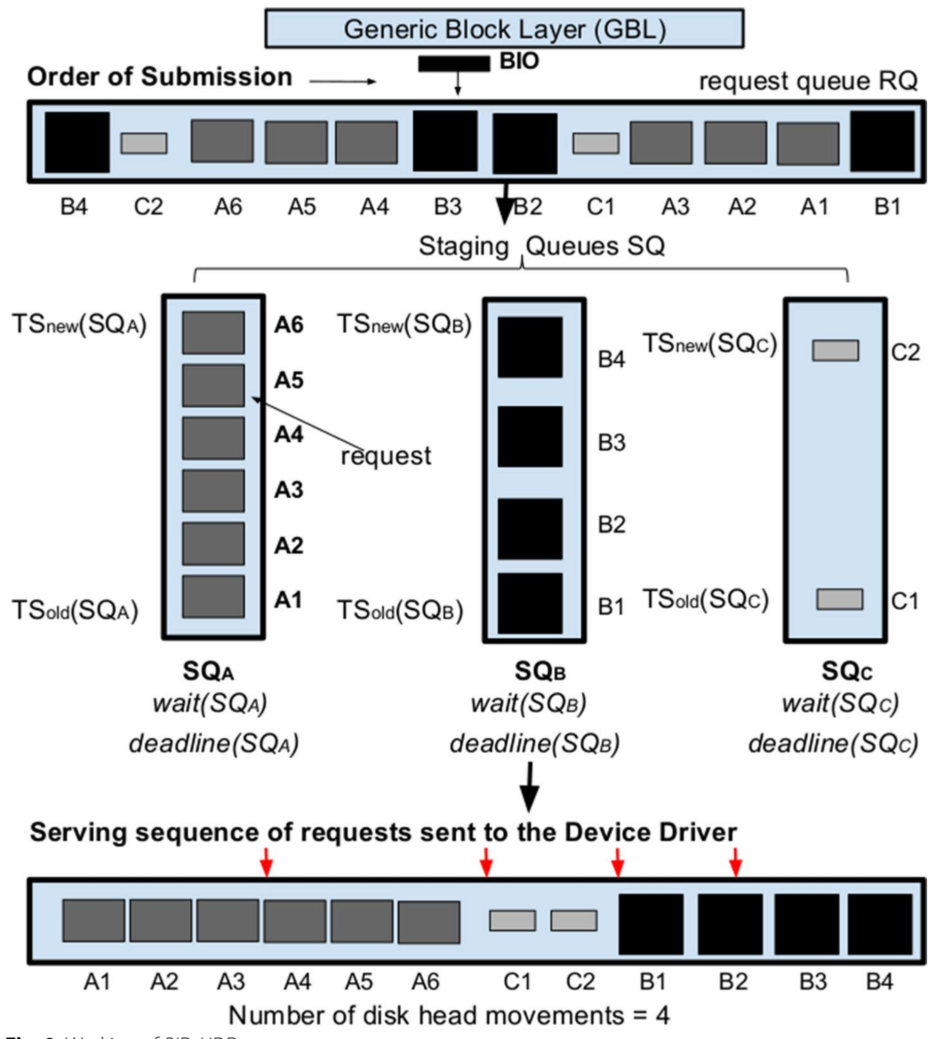

Generic Block Layer (GBL)
BIO
Order of Submission
request queue RQ
B4 C2 A6 A5 A4 B3 B2 C1 A3 A2 A1 B1
Staging Queues SQ
TSnew(SQA)
TSold(SQA)
A6 A5 A4 A3 A2 A1
request
TSnew(SQB)
TSold(SQB)
B4 B3 B2 B1
TSnew(SQC)
TSold(SQC)
C2 C1
SQA wait(SQA) deadline(SQA)
SQB wait(SQB) deadline(SQB)
SQC wait(SQC) deadline(SQC)
Serving sequence of requests sent to the Device Driver
A1 A2 A3 A4 A5 A6 C1 C2 B1 B2 B3 B4
Number of disk head movements = 4

Fig. 8 Working of *BID-HDD*

ALGORITHM 1: Stage Requests

```
for every process $p \in P$ 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 $deadline(SQ_p) = 500ms$;
```

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 "Hadoop MapReduce: working and workload characteristics" and "Requirements from a block I/O scheduler in Big Data deployments" sections, 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.

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 5000 ms. Whenever a blocking I/O (read) request is enqueued to SQ_p, the $deadline(SQ_p)$ is set to 500 ms if its current value is more than 500 ms. 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).

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 $deadline(SQ_p)$;
```

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 Fig. 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.).

BID-hybrid: contention avoidance utilizing multi-tier architecture

Due to physical limitation of HDDs, there have been recent efforts to incorporate flash based high-speed, non-volatile secondary memory devices, known as 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 "Secondary storage (block device) characteristics" section [1, 9].

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 [1, 3, 12, 25].

Data centers consists of many tiers of storage devices. All storage devices of the same type form a *tier* [21]. 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* [3, 4] is the automatic classification of how data is managed between heterogeneous tiers of storage in enterprise data-center environment [33]. 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 SDS solution.

Initial tier placement and BID-hybrid Our 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.

Majority of the existing literature (refer to "Related works" section) take *"tier placement"* decisions based on identification of randomness, i.e. deviation of LBA from that of the maximum group of requests in the block device request queue. The random blocks are tiered to a non-volatile storage device such as SSD. Tiering decisions based on deviation of LBA might be beneficial is some cases, such as those processes which might not exhibit sequentiality, but this would also result in unnecessary deportation to a faster tier. For example, multiple MapReduce processes can contend for the different regions of the same HDD. Therefore, at a single instance of time, in the request queue majority of the write requests might belong to one only process, while the second process which is also sequential might be able to submit few (tail-ending) requests due to request queue size (or CPU locking, blocking, etc.). In such a case these blocks might appear random while they are actually sequential. The implications of deportation might be more pronounced, due to additional unnecessary book-keeping overheads, consumption of limited write-erase cycles of SSDs, and, data management overheads, etc.

However, *BID-Hybrid* is designed to suit such 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 [10]) 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 [8, 28, 34]. 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.

Figure 9 shows the architecture of BID-Hybrid in multi-tier storage environments with the help of the I/O submission order as in Table 1. To show the effectiveness in offloading non-bulky requests to SSD during initial write, we change the type of operation from "reads" to "writes" of requests *C*1, *C*2 belonging to process *C*. In Fig. 9, we have shown the VFS, Volume Manager, Mapping Layer and VFS-SSD 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 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 VFS for infrastructural support for offloading and locating tiered SSD blocks. The working of BID-Hybrid is described as follows:

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

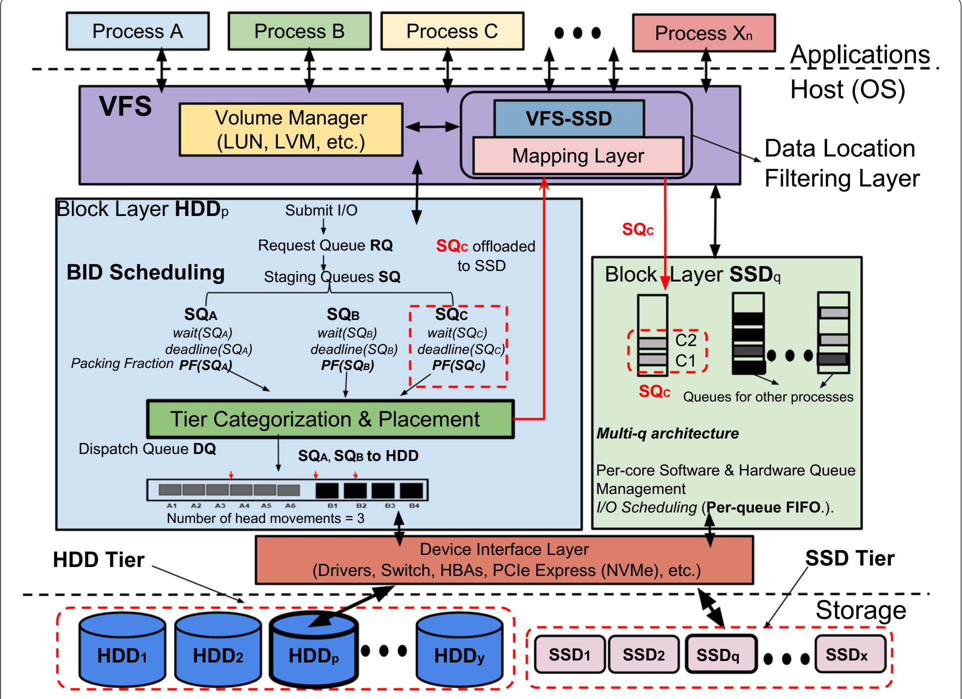


Fig. 9 Working architecture of *BID-Hybrid*

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.

Tier categorization 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 "BID-HDD: contention avoiding I/O scheduling for HDDs" section and Algorithm 1). Once the staging queue SQ_P is marked for flushing, the tier placement and 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 and categorization decision and dispatch of I/Os follows Algorithm 3.

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

$$\textit{Cumulative } I/O \textit{ RequestSize}_{write}(SQ_P) = \sum_{i=1}^{i=k} \textit{size of write request}_i(SQ_P)$$

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.) [23, 28]. 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 [23].

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. [35].

We have considered the case where dedicated SSDs 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. [10] employing a FIFO per queue scheduling scheme.

Consider Fig. 9, 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 [10] 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.

Experiments and performance evaluation

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

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 [36] and TPC Express Benchmark (TPCx-HS)-Hadoop suite [37]. 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 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 Fig. 10, 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 and 250 GB Serial ATA (SATA) HDD.

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* [38] 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, 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 and minimize the effects of the SCSI bus [34, 38, 39].

Table 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

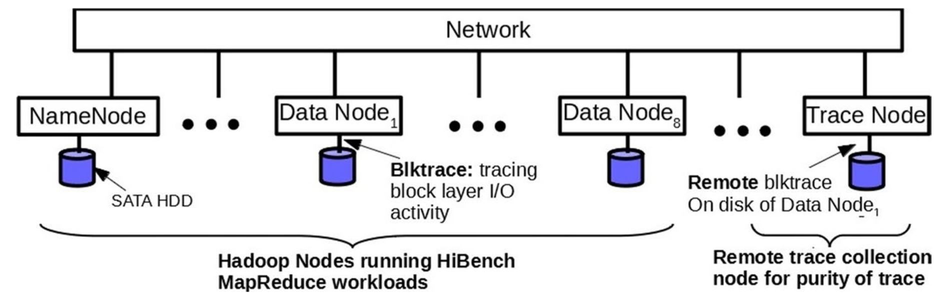

Network
NameNode
Data Node$_1$
Data Node$_8$
Trace Node
SATA HDD
Blktrace: tracing block layer I/O activity
Remote blktrace On disk of Data Node$_1$
Hadoop Nodes running HiBench MapReduce workloads
Remote trace collection node for purity of trace

Fig. 10 Experimental testbed: Hadoop cluster and capturing block layer I/O activity using *blktrace*

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 "Testbed: emulating cloud Hadoop workloads and capturing block layer activities" section) for application I/O submission order. The Simulator has three major modules: (a) *VFS Simulator* performs VFS 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 the components (see Fig. 11) 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 collected (refer to "Testbed: emulating cloud Hadoop workloads and capturing block layer activities" section). 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

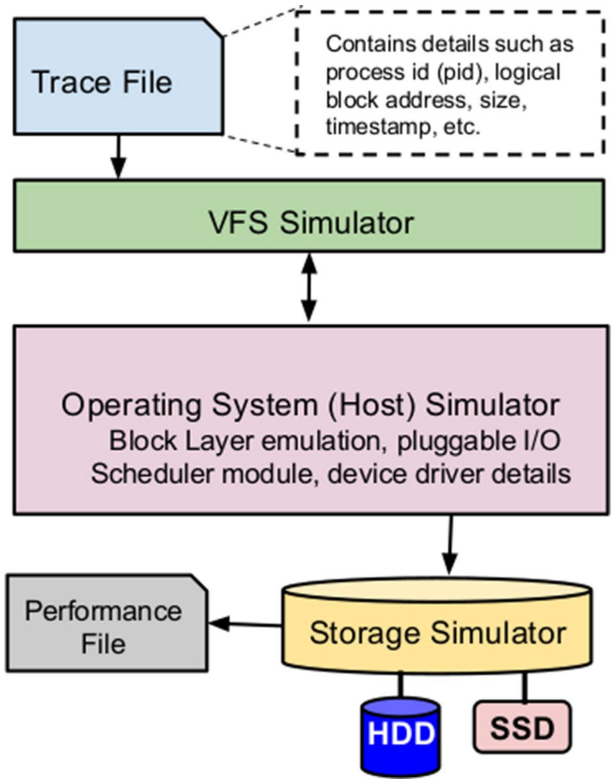

Trace File
Contains details such as process id (pid), logical block address, size, timestamp, etc.
VFS Simulator
Operating System (Host) Simulator
Block Layer emulation, pluggable I/O Scheduler module, device driver details
Performance File
Storage Simulator
HDD
SSD

Fig. 11 Simulator components

queue size and block device driver parameters. 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 [10]. 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 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 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. For SCMs (SSDs), the drive parameters include the no. of pages per block, size of each page, seek time (read, writes, erase) etc. The Storage Simulator is CHS compliant for 48-bit LBA. The Storage 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 [17, 40, 41]. 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.

Performance evaluation: results and discussions

We compare the effectiveness of our Contention Avoidance or I/O Scheduling schemes, BID-HDD and BID-Hybrid, 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 [6, 28]. Deadline I/O Scheduling leads to reduced throughput and result in large number of seeks [2, 6] 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 "Issues with current I/O schedulers" section). Hence, we compare our solutions with CFQ and Noop.

For our experiments, we use the default parameters as shown in Table 3, which is based on the storage devices and driver specifications.

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

Cumulative I/O completion time

Figure 12 represents the cumulative time taken (x-axis) by the block device (in case of Hybrid approach, devices) 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 12 demonstrates that BID-HDD outperforms CFQ and Noop for all the workloads. The savings in cumulative I/O completion time is maximum for *WordStandard-Deviation* & *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.

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

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

Table 3 Block device parameters in use for performance 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 × 4K pages
	1 page (bio vec) = 8 × 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 = 4 kB = access granularity
	Access time: read = 0.025 ms; write = 0.5 ms
	Specification based on SLC Flash SSD [9]

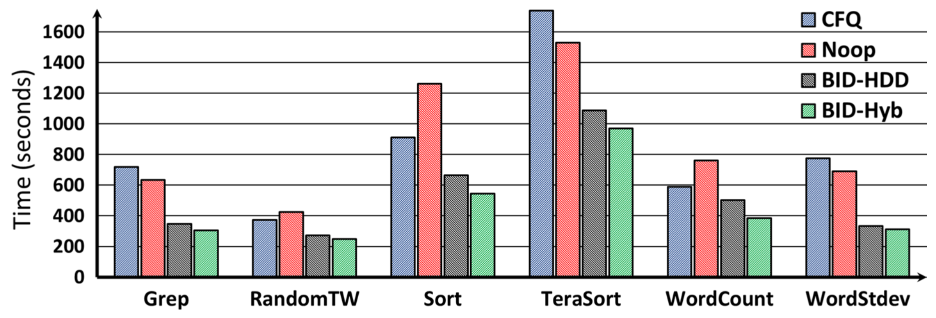


Fig. 12 Cumulative I/O completion time

of being fair to all contending processes (in terms of I/O bandwidth), multiplexes the requests. This nature of CFQ is evident from Fig. 13, 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 13b, c, 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 Fig. 13a, c, 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 Fig. 14, which shows the magnified view of Fig. 13a–c 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 Fig. 14, 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

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

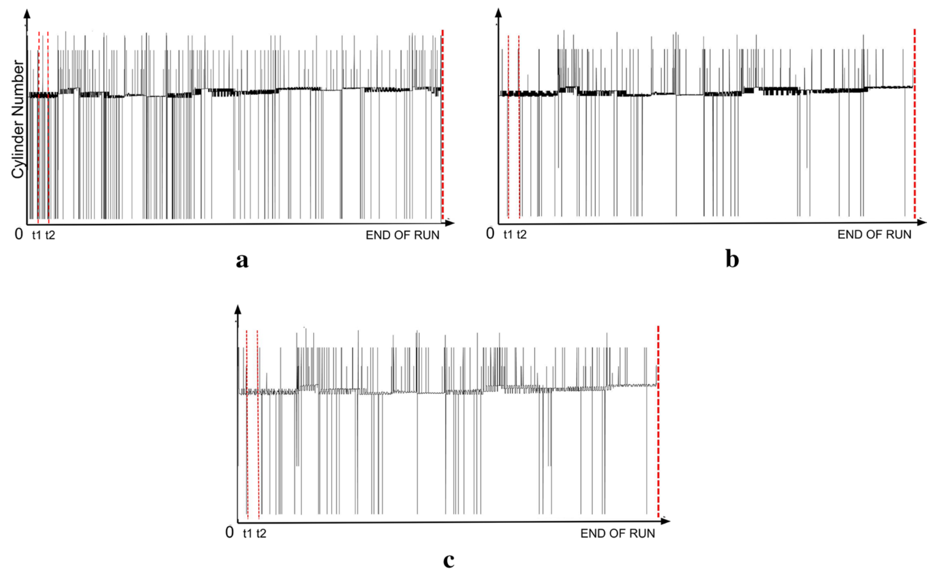


Fig. 13 Disk arm movements for *WordStandardDeviation* workload. **a** CFQ, **b** Noop, **c** BID-HDD

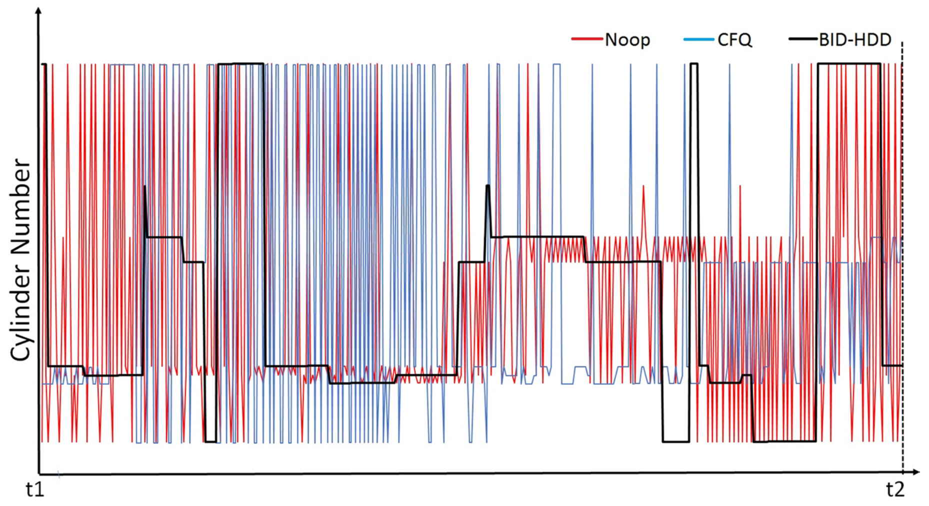


Fig. 14 Disk head movements for Noop, CFQ and BID-HDD between timestamps t_1 and t_2 for *WordStandard-Deviation*

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.

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 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 [10] 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 [8, 28, 34]. This in turn would lead to further smoothening of the BID-HDD graph of Fig. 14, with spikes in the graph being leveled.

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 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 processes 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 [5, 8, 10].

Takeaway 3 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.

Number of disk arm movements

Figure 15 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 "Cumulative I/O completion time" section and Fig. 14. 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.

In BID-Hybrid solution, 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.

Please note that Fig. 15 is the cumulative representation of information shown in Fig. 13. 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.

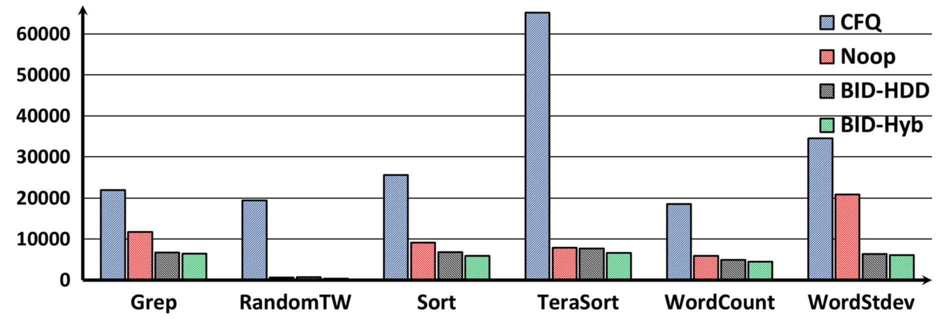


Fig. 15 Total number of disk arm movements

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

Disk head movement (seek) distance

Figure 16 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 Fig. 15), 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 ith 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 17 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.

For example, consider the I/O submission order in Table 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}.

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

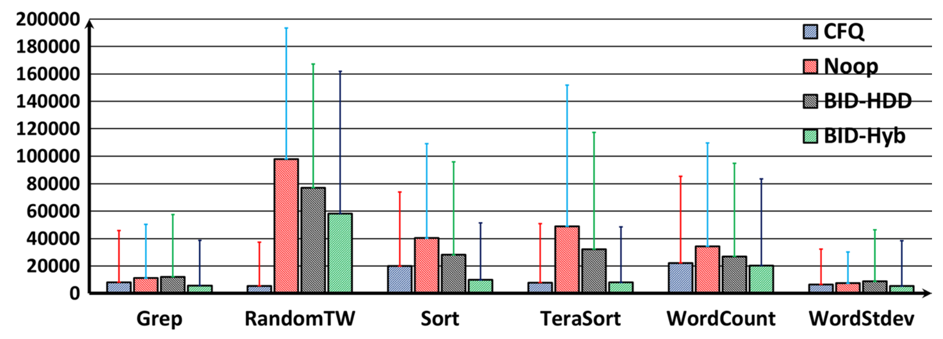


Fig. 16 Avg distance (no. of cylinders or tracks) per disk arm movement

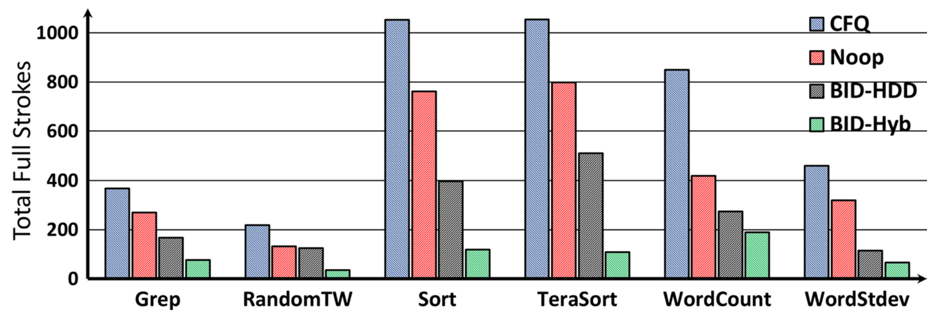


Fig. 17 Cumulative disk head movement distance

$$No.\ of\ head\ movements\ \text{“}n\text{”} = 8$$
$$AD_{seek} = \frac{TMD_{seek}}{n} = \frac{203}{8} = 25.37$$
$$Total\ No.\ of\ tracks\ (or\ cylinders)\ \text{“}Tr_{HDD}\text{”} = 100$$
$$TD_{sweep} = \frac{TMD_{seek}}{Tr_{HDD}} = \frac{203}{100} = 2.03$$

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 Fig. 17). Figure 17 shows the overall impact of employing a scheduler. It also shows that both, BID-HDD and BID-Hybrid drastically reduce the total distance traversed by the disk arm. Similar justification of the amortization effect in BID schemes, 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, Fig. 12 and total distance covered Fig. 17 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 schemes with disk optimizations schemes like Borg [8]. The sparse locality of data belonging to different applications can be optimized by employing self-optimizing block reorganizing solutions like Borg [8]. 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.

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 [3, 5, 8, 10]. 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.

Figures 18 and 19 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

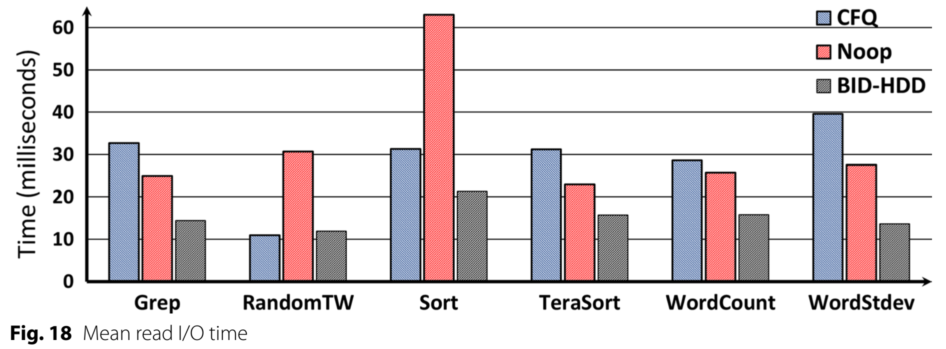


Fig. 18 Mean read I/O time

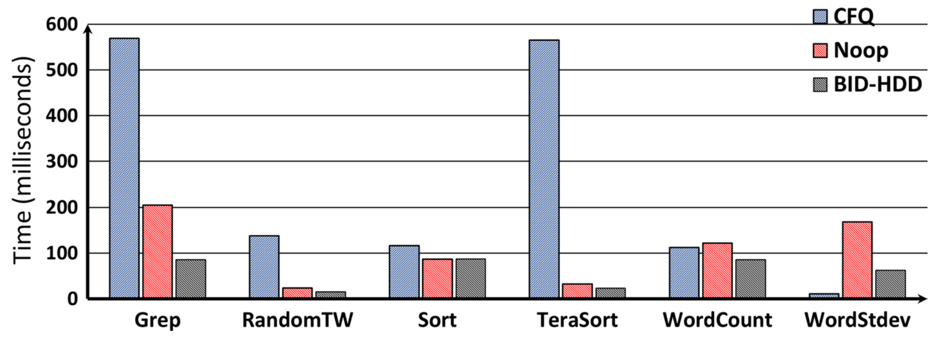


Fig. 19 Mean write I/O time

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.

BID is aimed to avoid contention following system constraints without compromising SLAs, as described in "Requirements from a block I/O scheduler in Big Data deployments" section. 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. 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.

Related works

The domain of storage technologies has been an active field of research. More recently, there have been research inclination in developing both, the software as well as physical architecture of NVMe, referred to as SCMs [3, 9, 12] to meet the SLAs of Big Data. We broadly classify the literature in our focus into: (a) block layer developments, mostly I/O Scheduling, and (b) multi-tier storage environment. Table 4 mentions state-of-the-art solutions in both these classifications.

Block layer developments, mostly I/O scheduling

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 [41], though we will limit our discussion to those approaches which are relevant to recent

Table 4 Related works categorization

Block layer (I/O scheduling)	Multi-tier
BID [42], Multi-Q [10], CFFQ [6] , SLASSD [7], Axboe [44], Hystor [34], PDC [43], ParDispatcher[49], BFQ [50], FlexDrive [12], AD [28], Borg [8]	ADLAM [33], SUORA [4], hatS [21], PDC [43], HRO [45], RPAC [46], RAF [47], PASS [48], Triple-H [11], ExaPlan [20], HybridStore [51], Hystor [34], Scarlett [52], DUX [1]

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 [10, 39] to cater the requirements of Big Data. Riska et al. [39] 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 [44] provides a brief overview of the Linux block layer, basic I/O units, request queue processing, etc. AD [28] 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. BORG [8] is a self-optimizing HDD based solution which re-organizes blocks in the block layer by forming sequences via calculating correlation amongst LBA 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.

Multi-q [10] 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. FlexDrive [12] 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.

CFFQ [6] 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. SLASSD [7] and Kim et al. [2] propose to ensure diverse SLAs, including reservations, limitations, and proportional sharing by their I/O Scheduling schemes in shared VM environment for SSDs. While SLASSD [7] uses an opportunistic goal oriented block I/O scheduling algorithm, Kim et al. [2] proposes host level SSD I/O schedulers, which are extensions of state-of-the-art I/O scheduling scheme CFQ. ParDispatcher [49] tries to utilize the parallelism in SSDs, by dividing the entire SSD into sub-regions, each having a different queue for dispatching requests. ParDipatcher might be good in applications which have more random I/Os otherwise, leading to increasing wait queues for popular sub-regions and bias in performance.

Multi-tier storage

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 [3, 12]. Though hard drives wont be replaced by NVMe devices (SSDs) in the near future, more prominently due to SSD's high TCO (cost/GB, write amplification, lifespan) [24], lack of consistent software stack (fabrics, interface and media characteristics) as well as non-uniform workload performance characteristics [1, 3, 9] (refer to "Secondary storage (block device) characteristics" section). 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 [4, 23]. 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 70s and 80s [53, 54]. 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 [53–55], therefore extensive work has been done by the database community [56–64].

Navathe [56], Cornell [59], March [57] and Chu [58] have been one the earliest studies in the filed of partitioning of databases. Navathe [56] 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 [58] developed an optimization model for minimizing overall costs by constricting response time and capacity with fixed number of copies of each file fragment. Cornell [59] 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 [57] 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 [53, 54]. Dynamically adaptive variations of these concepts have been explored thoroughly in the design of modern datastores [53, 54, 61]. These methods are used in online data partitioning such as O2P, H20 [60], etc., and disk based analytical databases [54, 63, 65]. In the Big Data ecosystem, most prominently the concepts of [56–59] have laid strong footing for the data layout design on HDFS [54, 65, 66]. Another use case has been in tuning of data stores [53, 67], 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 [55] have led to the design of modern Column stores and NoSQL datastores, most popularly, Hbase [23], WideTable [54], RCFile [65], 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 multi-tier storage solutions in literature is concentrated in 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 [23, 29], therefore faster SCMs (SSDs) are used as cache. ADLAM [33] proposes an adaptable data migration model based on the heat of data to determine the next hot data. HRO [45] 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. PDC [43] 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. RPAC [46] 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 (see "Related works" section) have been proposed, in shared Big Data cloud deployments due to the highly skewed, non-uniform and multiplexing workloads [28], prediction of utility value of blocks for tiering based on heat of data might not be a viable option.

However our proposed solution, BID-Hybrid 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 [8, 28, 34].

In the existing literature tiering is based on randomness in I/O to be defined as mere deviation of LBA. An application could be sequential but due to contention at the request queue to submit requests, could 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. ExaPlan [20] 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. Hybrid-Store [51] proposes a tool for improving capacity planning within cost-budgets and performance guarantees during deviations from expected workloads. DUX [1] 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.

OD [68] 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 RAF [47], 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 is read cache. Migration follows LFU, when the utilization rate of read cache is 90%. HatS [21] 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. Triple-H [11] 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 priority of usage.

Conclusion and future works

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. *BID-HDD*, which is essentially a block I/O scheduling scheme for disk based storage, results in 28–52% lesser time for all I/O requests than the best performing Linux disk schedulers. *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 6–23% performance gains over BID-HDD and 33–54% 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 [8], adaptive data migration policies of ADLAM [33], and replication-management of such as Triple-H [11]. This could solve the issue of workload and cost-aware tiering for large scale data-centers experiencing Big Data workloads.

Broader impact of this research would aid Data Centers in achieving their SLAs as well keeping the 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.

Authors' contributions
PM has designed, developed and implemented the idea and the experiments, analyzed the results and wrote the manuscript. AKS has guided and supervised this research throughout from the very beginning. Both authors read and approved the final manuscript.

Authors' information
Pratik Mishra is a Ph.D. Candidate at Iowa State University, Ames, Iowa, USA. His research work majorly involves designing (re-designing), and developing solutions in the Hadoop Storage Stack for inching towards the realization of Software Defined Data Centers. In the industry, he has worked with storage technologies in Hadoop, HBase and other Big Data Environments as an R&D Cloud/Big Data Intern-Firmware Engineer (2015 and 2016) at Seagate Technologies, USA. During his undergraduate, he worked as a Big Data (Social Network Data) Analytics Intern (2011) at Wipro Technologies and Visiting Summer Intern (2012) at Iowa State University, USA. He has published several research papers and book chapters. He has also served as a technical paper reviewer for reputed conferences and journals in cloud computing, distributed systems and big data systems.

Dr. Arun K. Somani is currently serving as Associate Dean for Research for College of Engineering and Anson Marston Distinguished Professor of Electrical and Computer Engineering at Iowa State University. Professor Somani's research interests are in the areas of dependable and high-performance system design, algorithms, and architecture, wavelength-division multiplexing-based optical networking, and image-based navigation techniques. He has published more than 300 technical papers, several book chapters, and one book, and has supervised more than 70 MS and more than 35 Ph.D. students. His research has been supported by several NSF-, industry- and DARPA-funded projects. He was the lead designer of an anti-submarine warfare system for the Indian navy, Meshkin fault-tolerant computer system architecture for the Boeing Company, Proteus multi-computer cluster-based system for US Coastal Navy, and HIMAP design tool for the Boeing Commercial Company. He was awarded Distinguished Engineer member grade of ACM in 2006, and elected Fellow of IEEE in 1999 for his contributions to "theory and applications of computer networks." He also is elected as a Fellow of AAAS in 2012.

Acknowledgements
An earlier version of this article has been published in the Proceedings of 24th IEEE Modeling, Analysis and Simulation of Computer and Telecommunication Systems "MASCOTS" 2016, London, UK as "Bulk I/O Storage Management for Big Data Applications" [42]. We thank Dr. Mayank Mishra, Post Doctoral Fellow at Iowa State University. He made several editorial comments and suggestions, held valuable discussion and provided feedback on design of experiments on a shorter version of this paper [42], where he was also a coauthor.

Competing interests
The authors declare that they have no competing interests.

Availability of data and materials
The trace input data is in hundreds of GBs and can be obtained using the open-source Linux utility *blktrace*, while running the cloud emulating Big Data benchmarks [36, 37] in a Hadoop deployment and following the instructions as discussed in "Testbed: emulating cloud Hadoop workloads and capturing block layer activities" section.

Funding
Research funded in part by Philip and Virginia Sproul Professorship Endowment at Iowa State University. The research reported in this paper also is partially supported by the HPC@ISU equipment at Iowa State University, some of which has been purchased through funding provided by NSF under MRI Grant Number CNS 1229081 and CRI Grant Number 1205413. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Appendix: Block I/O Kernel sub-structures

The generic block layer converts I/O requests to I/O operations known as block I/O (or BIO) structures (refer to "Background" section). The BIO data-structure are contiguous disk blocks and contain information such as type of operation (read/write), 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 block I/O scheduler is responsible for creation and merging of "request" structures from BIO structures as well as management of the "request queue".

Figure 20 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 [69], each "bio vec" represents a page in memory, by default it is 4096 bytes. Each BIO structure can contain a maximum of 256 [69] "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 [31]. The request queue length is limited by the

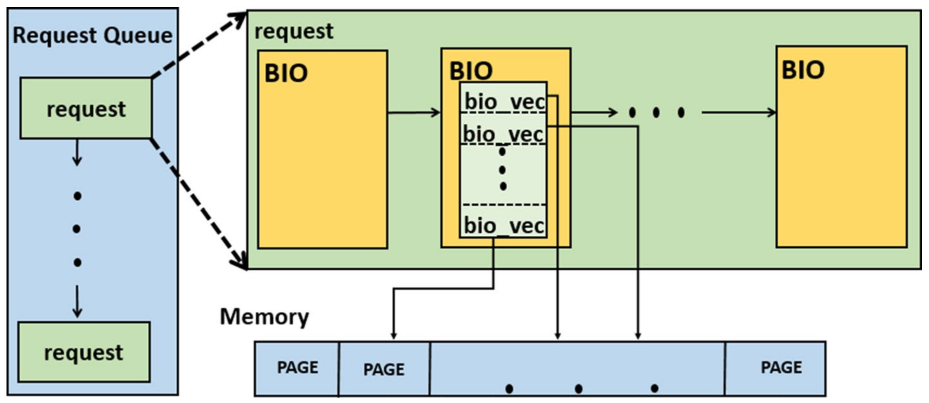


Fig. 20 Request queue processing structures

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) [31].

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 17 April 2017 Accepted: 8 June 2017
Published online: 19 June 2017

References

1. Krish K, Wadhwa B, Iqbal MS, Rafique MM, Butt AR. On efficient hierarchical storage for big data processing. In: 2016 16th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid). New York: IEEE; 2016. p. 403–8.
2. Kim J, Lee E, Noh SH. I/o scheduling schemes for better i/o proportionality on flash-based ssds. In: 2016 IEEE 24th international symposium on modeling, analysis and simulation of computer and telecommunication systems (MASCOTS). New York: IEEE; 2016. p. 221–30.
3. Nanavati M, Schwarzkopf M, Wires J, Warfield A. Non-volatile storage. Queue. 2015;13(9):20–332056.
4. Zhou J, Xie W, Noble J, Echo K, Chen Y. Suora: a scalable and uniform data distribution algorithm for heterogeneous storage systems. In: 2016 IEEE international conference on networking, architecture and storage (NAS). New York: IEEE; 2016. p. 1–10.
5. Joo Y, Park S, Bahn H. Exploiting i/o reordering and i/o interleaving to improve application launch performance. ACM Trans Storage. 2017;13(1):8–1817.
6. Yi M, Lee M, Eom YI. 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. New York: ACM; 2017. p. 87–1876. http://doi.acm.org/10.1145/3022227.3022313.
7. Park H, Yoo S, Hong C-H, Yoo C. Storage SLA guarantee with novel ssd i/o scheduler in virtualized data centers. IEEE Trans Parallel Distrib Syst. 2016;27(8):2422–34.
8. Bhadkamkar M, Guerra J, Useche L, Burnett S, Liptak J, Rangaswami R, Hristidis V. Borg: block-reorganization for self-optimizing storage systems. In: Proceedings of the 7th conference on file and storage technologies. FAST '09. pp. 183–196. Berkeley: USENIX Association; 2009. http://dl.acm.org/citation.cfm?id=1525908.1525922. Accessed 31 Mar 2017.
9. Mittal S, Vetter JS. A survey of software techniques for using non-volatile memories for storage and main memory systems. IEEE Trans Parallel Distrib Syst. 2016;27(5):1537–50.
10. Bjørling M, Axboe J, Nellans D, Bonnet P. Linux block io: introducing multi-queue ssd access on multi-core systems. In: Proceedings of the 6th international systems and storage conference. SYSTOR '13. New York: ACM; 2013. p. 22–12210. http://doi.acm.org/10.1145/2485732.2485740.
11. Islam NS, Lu X, Wasi-ur-Rahman M, Shankar D, Panda DK. Triple-h: a hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture. In: 2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid). 2015. p. 101–10.
12. Malladi KT, Awasthi M, Zheng H. Flexdrive: a framework to explore nvme storage solutions. In: 2016 IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on

smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS). New York: IEEE; 2016. p. 1115–22.
13. Love R. Linux Kernel development. 2010. p. 1–300. https://rlove.org/. Accessed 31 Mar 2017.
14. Avanzini A. Debugging Fanatic, Linux and Xen enthusiast. BFQ I/O scheduler. http://ari-ava.blogspot.com/2014/06/opw-linux-block-io-layer-part-1-base.html. Accessed 15 Apr 2016.
15. Vangoor BKR, Tarasov V, Zadok E. To fuse or not to fuse: performance of user-space file systems. In: Proceedings of FAST'17: 15th USENIX conference on file and storage technologies. 2017. p. 59.
16. Aghayev A, Ts'o T, Gibson G, Desnoyers P. Evolving ext4 for shingled disks. 2017.
17. Arpaci-Dusseau RH, Arpaci-Dusseau AC. Operating systems: three easy pieces, vol. 151. 2014.
18. Zheng D, Burns R, Szalay AS. 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. New York: ACM; 2013. p. 69.
19. Moon S, Lee J, Sun X, Kee Y-S. Optimizing the hadoop MapReduce framework with high-performance storage devices. J Supercomput. 2015;71(9):3525–48.
20. Iliadis I, Jelitto J, Kim Y, Sarafijanovic S, Venkatesan V. Exaplan: queueing-based data placement and provisioning for large tiered storage systems. In: 2015 IEEE 23rd international symposium on modeling, analysis and simulation of computer and telecommunication systems (MASCOTS). 2015. p. 218–27.
21. Krish KR, Anwar A, Butt AR. Hats: a heterogeneity-aware tiered storage for hadoop. In: 2014 14th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid). 2014. p. 502–11.
22. Eshghi K, Micheloni R. Ssd architecture and pci express interface. In: Inside solid state drives (SSDs). 2013. p. 19–45. https://link.springer.com/chapter/10.1007/978-94-007-5146-0_2.
23. Harter T, Borthakur D, Dong S, Aiyer A, Tang L, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Analysis of hdfs under hbase: a facebook messages case study. In: Proceedings of the 12th USENIX conference on file and storage technologies (FAST 14). Santa Clara: USENIX; 2014. p. 199–212. https://www.usenix.org/conference/fast14/technical-sessions/presentation/harter. Accessed 31 Mar 2017.
24. Yang Y, Zhu J. Write skew and zipf distribution: evidence and implications. Trans Storage. 2016;12(4):21–12119.
25. Roussos K. Storage virtualization gets smart. Queue. 2007;5(6):38–44.
26. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM. 2008;51:107–13.
27. White T. Hadoop: the definitive guide. 2012. p. 1–684. http://hadoopbook.com/. Accessed 31 Mar 2017.
28. Ibrahim S, Jin H, Lu L, He B, Wu S. Adaptive disk i/o scheduling for MapReduce in virtualized environment. In: 2011 international conference on parallel processing. 2011. p. 335–44.
29. Krish K, Khasymski A, Butt AR, Tiwari S, Bhandarkar M. Aptstore: dynamic storage management for hadoop. In: 2013 IEEE 5th international conference on cloud computing technology and science (CloudCom), vol. 1. New York: IEEE; 2013. p. 33–41.
30. Schönberger G. Linux I/O scheduler, Thomas Krenn. https://www.thomas-krenn.com/de/wiki/Linux_I/O_Scheduler.
31. Inc., R.H. Linux performance tuning guide. Red-hat Enterprise
32. Seelam S, Romero R, Teller P, Buros B. Enhancements to linux i/o scheduling. In: Proc. of the Linux symposium, vol. 2. 2005. p. 175–92.
33. Zhang G, Chiu L, Liu L. Adaptive data migration in multi-tiered storage based cloud environment. In: 2010 IEEE 3rd international conference on cloud computing (CLOUD). New York: IEEE; 2010. p. 148–55.
34. Chen F, Koufaty DA, Zhang X. Hystor: making the best use of solid state drives in high performance storage systems. In: Proceedings of the international conference on supercomputing. New York: ACM; 2011. p. 22–32. http://doi.acm.org/10.1145/1995896.1995902.
35. Pfefferle J, Stuedi P, Trivedi A, Metzler B, Koltsidas I, Gross TR. A hybrid i/o virtualization framework for rdma-capable network interfaces. In: ACM SIGPLAN notices, vol. 50. New York: ACM; 2015. p. 17–30.
36. Huang S, Huang J, Dai J, Xie T, Huang B. The hibench benchmark suite: characterization of the MapReduce-based data analysis. In: 2010 IEEE 26th international conference on data engineering workshops (ICDEW). 2010. p. 41–51.
37. TPC(tm). TPC Express Benchmark(tm) HS (TPCx-HS)-Hadoop suite overview. http://www.tpc.org/tpcx-hs/. Accessed 31 Mar 2017.
38. Brunelle AD. blktrace user guide. 2007.
39. Riska A, Larkby-Lahet J, Riedel E. Evaluating block-level optimization through the io path. In: 2007 USENIX annual technical conference on proceedings of the USENIX annual technical conference. ATC'07. Berkeley: USENIX Association; 2007. p. 19–11914. http://dl.acm.org/citation.cfm?id=1364385.1364404. Accessed 31 Mar 2017.
40. Bian H, Yan Y, Tao W, Chen LJ, Chen Y, Du X, Moscibroda T. Wide table layout optimization based on column ordering and duplication. In: Proceedings of the 2017 ACM international conference on management of data. New York: ACM; 2017. p. 299–314.
41. Ruemmler C, Wilkes J. An introduction to disk drive modeling. Computer. 1994;27(3):17–28.
42. Mishra P, Mishra M, Somani AK. Bulk i/o storage management for big data applications. In: 2016 IEEE 24th international symposium on modeling, analysis and simulation of computer and telecommunication systems (MASCOTS). New York: IEEE; 2016. p. 412–7.
43. Chang H-P, Liao S-Y, Chang D-W, Chen G-W. Profit data caching and hybrid disk-aware completely fair queuing scheduling algorithms for hybrid disks. Softw Pract Exp. 2015;45(9):1229–49.
44. Axboe J. Linux block io—present and future. In: Ottawa Linux Symp. 2004. p. 51–61.
45. Lin L, Zhu Y, Yue J, Cai Z, Segee B. 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. 2011. p. 318–25.
46. Ye F, Chen J, Fang X, Li J, Feng D. A regional popularity-aware cache replacement algorithm to improve the performance and lifetime of ssd-based disk cache. In: 2015 IEEE international conference on networking, architecture and storage (NAS). 2015. p. 45–53.
47. Liu Y, Huang J, Xie C, Cao Q. Raf: a random access first cache management to improve ssd-based disk cache. In: 2010 IEEE fifth international conference on networking, architecture and storage (NAS). 2010. p. 492–500.

48. Xiao W, Lei X, Li R, Park N, Lilja DJ. Pass: a hybrid storage system for performance-synchronization tradeoffs using ssds. In: 2012 IEEE 10th international symposium on parallel and distributed processing with applications. 2012. p. 403–10.
49. Wang H, Huang P, He S, Zhou K, Li C, He X. A novel i/o scheduler for ssd with improved performance and lifetime. In: 2013 IEEE 29th symposium on mass storage systems and technologies (MSST). New York: IEEE; 2013. p. 1–5.
50. Valente P, Andreolini M. Improving application responsiveness with the bfq disk i/o scheduler. In: Proceedings of the 5th annual international systems and storage conference. SYSTOR '12. New York: ACM; 2012. p. 6–1612. http://doi.acm.org/10.1145/2367589.2367590.
51. Kim Y, Gupta A, Urgaonkar B, Berman P, Sivasubramaniam A. 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. 2011. p. 227–36.
52. Ananthanarayanan G, Agarwal S, Kandula S, Greenberg A, Stoica I, Harlan D, Harris E. Scarlett: coping with skewed content popularity in MapReduce clusters. In: Proceedings of the sixth conference on computer systems. New York: ACM; 2011. p. 287–300.
53. Galaktionov V, Chernishev G, Smirnov K, Novikov B, Grigoriev DA. 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. Berlin: Springer; 2016. p. 163–77.
54. Li Y, Patel JM. Widetable: an accelerator for analytical data processing. Proc VLDB Endow. 2014;7(10):907–18.
55. Agrawal S, Narasayya V, Yang B. Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data. New York: ACM; 2004. p. 359–70.
56. Navathe S, Ceri S, Wiederhold G, Dou J. Vertical partitioning algorithms for database design. ACM Trans Database Syst. 1984;9(4):680–710.
57. March ST, Rho S. Allocating data and operations to nodes in distributed database design. IEEE Trans Knowl Data Eng. 1995;7(2):305–17.
58. Chu WW. Optimal file allocation in a multiple computer system. IEEE Trans Comput. 1969;100(10):885–9.
59. Cornell DW, Yu PS. An effective approach to vertical partitioning for physical design of relational databases. IEEE Trans Softw Eng. 1990;16(2):248–58.
60. Alagiannis I, Idreos S, Ailamaki A. H2o: a hands-free adaptive store. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. New York: ACM; 2014. p. 1103–14.
61. Curino C, Jones E, Zhang Y, Madden S. Schism: a workload-driven approach to database replication and partitioning. Proc VLDB Endow. 2010;3(1–2):48–57.
62. Jindal A, Dittrich J. Relax and let the database do the partitioning online. In: International workshop on business intelligence for the real-time enterprise. Berlin: Springer; 2011. p. 65–80.
63. Jindal A, Palatinus E, Pavlov V, Dittrich J. A comparison of knives for bread slicing. Proc VLDB Endow. 2013;6(6):361–72.
64. LeFevre J, Sankaranarayanan J, Hacigumus H, Tatemura J, Polyzotis N, Carey MJ. Miso: souping up big data query processing with a multistore system. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. New York: ACM; 2014. p. 1591–602.
65. He Y, Lee R, Huai Y, Shao Z, Jain N, Zhang X, Xu Z. Rcfile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: 2011 IEEE 27th international conference on data engineering (ICDE). New York: IEEE; 2011. p. 1199–208.
66. Jindal A, Quiané-Ruiz J-A, Dittrich J. Trojan data layouts: right shoes for a running elephant. In: Proceedings of the 2nd ACM symposium on cloud computing. New York: ACM; 2011. p. 21.
67. Guo S, Xiong J, Wang W, Lee R. Mastiff: a MapReduce-based system for time-based big data analytics. In: 2012 IEEE international conference on cluster computing (CLUSTER). New York: IEEE; 2012. p. 72–80.
68. Shi H, Arumugam RV, Foh CH, Khaing KK. Optimal disk storage allocation for multi-tier storage system. In: APMRC, 2012 digest. New York: IEEE; 2012. p. 1–7.
69. Linux Cross Reference Free Electrons, Embedded Linux Experts, Linux Kernel 4.15 Source Code. http://lxr.free-electrons.com/source/block/. Accessed 21 May 2016.