2018 IEEE International Conference on Cluster Computing

Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers

Anthony Kougkas, Hariharan Devarajan, Xian-He Sun, and Jay Lofstead*
Illinois Institute of Technology, Department of Computer Science, *Sandia National Laboratories
{akougkas, hdevarajan}@hawk.iit.edu, sun@iit.edu, gflofst@sandia.gov

***Abstract*—Modern HPC systems employ burst buffer installations to reduce the peak I/O requirements for external storage and deal with the burstiness of I/O in modern scientific applications. These I/O buffering resources are shared between multiple applications that run concurrently. This leads to severe performance degradation due to contention, a phenomenon called *cross-application I/O interference*. In this paper, we first explore the negative effects of interference at the burst buffer layer and we present two new metrics that can quantitatively describe the slowdown applications experience due to interference. We introduce *Harmonia*, a new dynamic I/O scheduler that is aware of interference, adapts to the underlying system, implements a new 2-way decision-making process and employs several scheduling policies to maximize the system efficiency and applications' performance. Our evaluation shows that Harmonia, through better I/O scheduling, can outperform by 3x existing state-of-the-art buffering management solutions and can lead to better resource utilization.**

***Index Terms*—Burst Buffers, I/O Scheduling, I/O Interference, Multi-Tenancy, I/O Policies, Shared Buffers**

I. Introduction

Modern HPC applications generate massive amounts of data. However, the improvement in the speed of disk-based storage systems has been much slower than that of memory, creating a significant I/O performance gap [1], [2]. In a large scale environment, the underlying file system is usually a remote parallel file system (PFS) with Lustre [3], GPFS [4], and PVFS2 [5] being some popular examples. However, as we move towards the exascale era, most of these file systems face significant challenges in performance, scalability, complexity, and limited metadata services [6], creating the so called *I/O bottleneck* which will lead to less scientific productivity [7], [8]. Non-Volatile Memory Express devices (NVMe), Solid-State Drives (SSD), and shared burst buffer (BB) nodes have been introduced to alleviate this issue [9], [10], [11]. Several new supercomputers employ such low-latency devices to deal with the burstiness of I/O [12], [13], reducing the peak I/O requirements for external storage [14]. For example, Cori system at the National Energy Research Scientific Computing Center (NERSC), uses CRAY's Datawarp technology [15]. Los Alamos National Laboratory Trinity supercomputer uses BBs with a 3.7 PB capacity and 3.3 TB/s bandwidth. Summit in Oak Ridge National Lab will also employ fast NVMe storage for buffering [16]. In fact, IDC reports [17] that as solid state devices become more affordable (2.2x price premium by 2021), storage systems will be equipped with SSD-only deployments making disk-based PFS archival in nature. As multiple layers of storage are introduced into HPC systems, the complexity of data movement among the layers increases significantly, making it harder to take advantage of the high-speed or low-latency storage systems [18].

Another characteristic of modern supercomputers that leads to challenges regarding I/O access is multiple concurrent jobs. Systems like Sunway TaihuLight, the top supercomputer in the Top500 list, have million of cores and run multiple applications concurrently [19]. Due to the sharing of resources such as compute nodes, networks, remote PFS, performance variability is observed [20]. This phenomenon is called *cross-application interference* and is common in most HPC sites [21]. The interference generally originates from concurrent access by multiple applications to shared resources. While computing and network resources can be shared effectively by state-of-the-art job schedulers, the same cannot be said about the storage resources. In fact, [22] and [23] suggest that I/O congestion, within and across independent jobs, is one of the main problems for future HPC machines. A lot of work has been done in PFS to mitigate the effects of I/O interference [19], [24], [25], [26], [27]. However, with the wider adoption of extra layers in the memory and storage hierarchy, such as BBs, extra care needs to be applied to coordinate the access to this new layer by multiple applications that shared it.

There are several characteristics of shared I/O buffering nodes that make scheduling I/O to this new layer quite challenging. First, BB nodes are placed either inside or very close to the compute-node network fabric. The lower access latency and the higher bandwidth of these networks change the way applications perform I/O compared to a traditional remote PFS that uses a slower network. Second, BBs can be used in several ways: as a cache on top of the PFS as a fast temporary storage for intermediate results or out-of-core applications (data may or may not need to be persisted), and as an in-situ/in-transit visualization and analysis. These use cases are fundamentally different from a file system where all I/O requests are persisted and strongly consistent. Lastly, BBs are presented to applications through reservations made to the central batch scheduler (e.g., Slurm [28]) whereas I/O access to PFS is performed via a mounting point and interleaved requests are serviced by the file system scheduler in a first-come-first-serve fashion. The above characteristics constitute traditional PFS I/O schedulers not suitable for this new storage layer and special attention in scheduling the I/O is needed to mitigate the negative effects of cross-application interference.

2168-9253/18/$31.00 ©2018 IEEE
DOI 10.1109/CLUSTER.2018.00046

IEEE computer society

In this paper, we propose Harmonia (greek word meaning "*in agreement or concord*"), a new dynamic I/O scheduler tailored for systems with shared I/O buffering nodes. Harmonia is a scheduler that is interference-aware, operates in a finer granularity, and is adaptive to the current system status. Our proposed 2-way design allows Harmonia to make scheduling decisions by collecting information from applications (i.e., intention to perform I/O) and the buffering nodes (i.e., available or busy device status) at the same time. We also present a novel metric, called *Medium Sensitivity to Concurrent Access* ($MSCA$), that captures how each type of buffering medium (i.e., NVMe, SSD, HDD) handles concurrent accesses. This metric helps Harmonia make better decisions when scheduling I/O by tuning the concurrency. We investigate how interference affects the application's execution time and we model this performance degradation by introducing a metric called *Interference Factor* (I_f). Harmonia uses a dynamic programming algorithm to optimize the scheduling of individual I/O phases on available buffers. By over-provisioning buffer nodes and by overlapping computation with I/O, Harmonia is able to reduce the scheduling cost in several metrics such as maximum bandwidth, minimum stall time, fairness, and buffer efficiency. Lastly, Harmonia offers the infrastructure to system administrators who can now offer *discounted* scheduling to their users. This effectively means that users will have the flexibility to "pay" (e.g., CPU hours or $ in reservation) according to their I/O needs. If an application is willing to allow its I/O to run longer, then Harmonia can "charge" this user less while still offering competitive performance. Scheduling discounts can easily be implemented by Harmonia's complete set of scheduling policies. The contributions of this work are:

- we introduce two new metrics that model performance degradation due to concurrent accesses and interference (i.e., resource contention), Section III.
- we present the design and implementation of a new BB I/O scheduler, called Harmonia, Section IV.
- we introduce three techniques to perform *I/O Phase Detection*, Section IV-A.
- we propose five new scheduling policies that optimize the performance of the buffering layer, Section IV-B.

II. Background and Motivation

Typically in HPC, storage systems are subjected to periodic, intense, short I/O phases (or bursts) that usually occur between computation phases and mostly consist of defensive I/O (i.e., simulation checkpoint) [29], [30], [31]. To better handle these I/O intensive phases, a new storage layer that sits between the application and the remote PFS has been introduced. This layer is commonly called *burst buffers* and its main functionality is to quickly absorb I/O requests from the application processing elements and asynchronously push them down to the PFS [32]. Burst buffers have evolved well beyond increasing the total I/O bandwidth available to applications and optimizing the I/O operations per second (i.e., IOPS) [14]. The addition of BBs to any computing infrastructure can make storage solution smarter and more efficient [33]. There are several BB deployments in supercomputers of today. These include, Cori at NERSC, Trinity at Los Alamos National Laboratory and KAUST at Shaheen Supercomputing Laboratory. All these use Cray's DataWarp Technology [15] for their BB management. NERSC demonstrated [34] an improvement of 60% performance on balanced usage over applications not using BB acceleration. However, they also stated that when two compute nodes share a BB node, their accesses compete for bandwidth which resulted in performance degradation for both jobs. This phenomenon is even stronger for data-intensive applications.

Burst buffers are proven to be valuable and efficient in accelerating the I/O but the software that manages this new layer is still new and not as mature as other traditional storage solutions such as PFS. We identify several challenges when scheduling I/O on shared buffering nodes [35]:

a) Reserving BB allocations through the main system batch scheduler does not account for interference on the buffering destination; if there is space on the buffer nodes' drives, the scheduler will place multiple applications on the same nodes which leads to performance degradation. The scheduler needs to be aware of concurrent accesses and avoid interference.

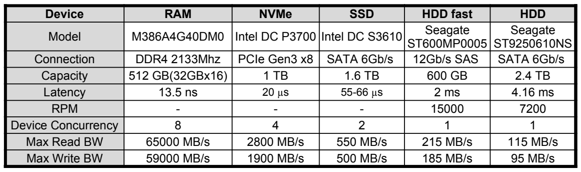
b) The lifetime of BB allocations is the same as the application that created them; this allocation exclusivity in terms of space and network leads to underutilization of the buffer resources since applications perform I/O in phases or bursts. The scheduler should be able to leverage the several computation phases that applications perform in between of I/O bursts and therefore could over-provision the buffer resources whenever possible (i.e., while one application is doing computation, another application can use the same buffers).

c) If the application uses more space than the allocation it has, the entire application will terminate unwillingly due to security and data consistency issues; this leads to users reserving more space than they actually need, and thus, waste of valuable resources. The scheduler should be able to dynamically extend temporarily the allocation space and continue execution.

d) When all buffer capacity is already allocated, any future application will wait in the queue of the system batch scheduler until resources become available again; this increases the scheduling time, even if there is plenty of unused space, leading to low system efficiency. The scheduler should take into account the type of reservations that are already allocated (i.e., some allocations are only for writing intermediate results and not for reading) and try to leverage any free buffer space.

e) When buffers are synchronizing data with the underlying parallel file system, any incoming I/O is severely stalled. For instance, when the buffer nodes are acting as the PFS cache, the application should be able to write data on the buffers while the buffers are flushing their contents on the PFS. The scheduler should be aware of the buffer nodes' status in order to smartly place incoming I/O to available nodes, and thus, provide performance guarantees to the application.

Understanding and solving those challenges motivates the proposal of Harmonia. Through our design choices, we aim to increase the buffering system efficiency, further accelerate applications' I/O by minimizing waiting times, mitigate



Device	RAM	NVMe	SSD	HDD fast	HDD
Model	M386A4G40DM0	Intel DC P3700	Intel DC S3610	Seagate ST600MP0005	Seagate ST9250610NS
Connection	DDR4 2133Mhz	PCIe Gen3 x8	SATA 6Gb/s	12Gb/s SAS	SATA 6Gb/s
Capacity	512 GB(32GBx16)	1 TB	1.6 TB	600 GB	2.4 TB
Latency	13.5 ns	20 µs	55-66 µs	2 ms	4.16 ms
RPM	-	-	-	15000	7200
Device Concurrency	8	4	2	1	1
Max Read BW	65000 MB/s	2800 MB/s	550 MB/s	215 MB/s	115 MB/s
Max Write BW	59000 MB/s	1900 MB/s	500 MB/s	185 MB/s	95 MB/s

Fig. 1. Storage Medium Specifications.

resource contention, and leverage the buffering medium's characteristics to maximize the overall system performance.

III. Interference Sensitivity

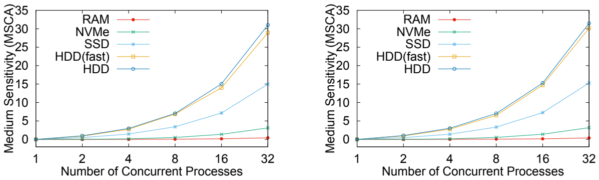
A. *Buffering Medium Sensitivity to Concurrent Accesses*

In systems equipped with I/O buffering capabilities, like supercomputers with burst buffer nodes, several devices such as RAM, SSDs, and HDDs can be used as the buffering destination. The advent of new memory-class storage mediums such as NVMe (i.e., non-volatile flash-based chips using new controllers such as NVMe) can lead to better I/O buffering due to lower latencies, higher bandwidth, and more medium concurrency. However, the burst buffer I/O scheduler needs to be aware of how each storage medium handles concurrent and interleaved requests coming from the application (*App_Concurrent_Accesses*). In order to determine the various performance variabilities these mediums demonstrate under concurrent access we introduce a new metric called *medium sensitivity to concurrent access* (MSCA). MSCA, shown in Eq.1, is defined as the rate at which each medium experiences a bandwidth reduction due to interference (i.e., concurrent accesses competing for the resource). The bandwidth reduction rate is linearly correlated to the *I/O device concurrency* which refers to hardware characteristics such as number of bus lanes, controller channels, memory banks (i.e., internal parallelism), and the maximum bandwidth each can achieve.

$$MSCA = \frac{App\_Concurrent\_Accesses}{I/O\_Device\_Concurrency} \times \frac{MaxBW - RealBW}{MaxBW} \quad (1)$$

Higher MSCA values mean that the medium is more sensitive to concurrent access. Harmonia takes this value into account to minimize the interference effect stemming from the underlying storage medium. To evaluate MSCA, we performed a series of tests on Chameleon testbed [36]. Specifically we used the bare metal configuration on the special storage hierarchy appliances. The specifications of each storage medium used in this evaluation can be seen in Figure 1. We mounted each storage device as an XFS Linux file system and we used a RamDisk (i.e., ramfs in main memory) as a reference. We bypassed the OS kernel buffering by using the O_DIRECT flag in the `open64()` and we flushed the OS cache before every run to make sure that I/O reaches the storage devices. Finally, we repeated the tests 10 times and reported the average.

As a driver program, we wrote a synthetic benchmark that stresses the storage medium with concurrent operations. In this benchmark, each process performs sequential I/O in a file-per-process pattern. The total I/O is 2GB and the request size is 64MB. We increased the number of MPI ranks that concurrently issue the I/O requests from 1 to 32 (weak scaling) and measured the achieved bandwidth. Note that the CPU has 48 cores. Figure 2 shows the results for write (a) and read (b) operations. As expected, RAM is the least sensitive buffering destination with a MSCA value of only 0.43 for 32 concurrent ranks since RAM has the most access lanes. On the other hand, traditional spinning hard drives, connected via a typical SATA III controller, have the highest MSCA value of around 30 for 32 concurrent ranks. This is reasonable since neither the controller nor the device itself provide more than one access lane. The SSD is also connected via a SATA III but the device we used has some degree of internal parallelism with two controller channels and memory banks being used at the same time. The MSCA value for SSD is close to 15 for 32 concurrent ranks which is half than that of the HDD. Finally, and most interestingly, NVMe demonstrated a MSCA value of around 3 placing it closer to RAM values. The reason that NVMe is less sensitive to concurrent accesses is that the NVMe controller uses the PCIe bus that provides 8 concurrent lanes. Note that MSCA does not describe how speedy a device is but rather how sensitive it is to concurrent accesses. All devices and controllers will eventually get saturated and the bandwidth that the application receives will drop. MSCA shows how quickly this happens for each type of buffering destination. This information is useful to Harmonia in order to make better scheduling decisions that take into account the characteristics of the buffering medium.



(a) Write operations

(b) Read operations

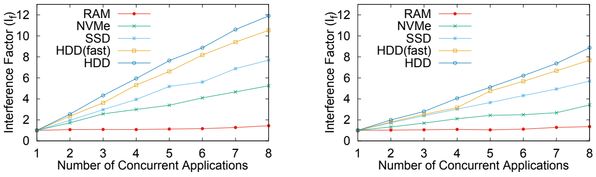
Fig. 2. Sensitivity to Concurrent Access (MSCA).

B. *Slowdown due to Cross-Application Interference*

When applications run concurrently in a system they compete for shared resources such as compute, network, and I/O. This phenomenon is called *cross-application interference* and can cause severe performance degradation. It is in fact one of the most complex manifestations of performance variability on large scale parallel computers [37]. In this subsection we focus on how this issue manifests when applications compete for access to shared buffering nodes. We leverage a metric we first introduced in our previous work [25] called *interference factor* and which can describe the slowdown applications experience due to resource contention. The interference factor is defined for a single application as the execution time with interference divided by the execution time without interference:

$$I_f = \frac{Execution\ Time_{with\_interference}}{Execution\ Time_{without\_interference}} \quad (2)$$

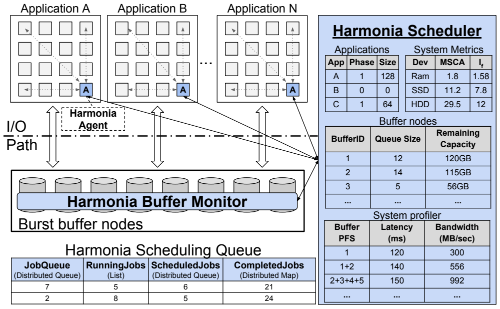
This metric is appropriate to study the effects of interference because it provides an absolute reference for a non-interfering system when $I_f = 1$. An interference factor of 2 means that



(a) Write operations

(b) Read operations

Fig. 3. Application Slowdown due to Interference (I_f).



Harmonia Scheduling Queue

JobQueue (Distributed Queue)	RunningJobs (List)	ScheduledJobs (Distributed Queue)	CompletedJobs (Distributed Map)
7	5	6	21
2	8	5	24

Applications

App	Phase	Size
A	1	128
B	0	0
C	1	64

System Metrics

Dev	MSCA	I_f
Ram	1.8	1.58
SSD	11.2	7.8
HDD	29.5	12

Buffer nodes

BufferID	Queue Size	Remaining Capacity
1	12	120GB
2	14	115GB
3	5	56GB
...	...	...

System profiler

Buffer PFS	Latency (ms)	Bandwidth (MB/sec)
1	120	300
1+2	140	556
2+3+4+5	150	992
...	...	...

Fig. 4. Harmonia high-level architecture.

the application experiences a 2x slowdown due to contention. Moreover, it allows the comparison of applications that have different size or different I/O requirements and is therefore a context-dependent measure. While cross-application interference has long been an important challenge in HPC systems, prior work focuses on different subsystems, such as compute and network, and has not taken into account the newly added burst buffer layer as well as new hardware storage devices.

To quantitatively measure the negative effects of cross-application interference, we performed the following motivating tests. We used the same machine and benchmark as the previous subsection III-A. We run multiple instances of the same benchmark at the same time, which introduces contention for the buffer resource. Figure 3 shows the results of scaling the test up to 8 concurrent instances expressed in terms of interference factor. As it can be seen, when 8 applications use RAM as their buffering destination the interference factor is around 1.45, for NVMe 5.24, for SSD 7.69, for the fast HDD 10.52, and for the typical HDD 11.90. Read operations in figure 3 (b) follow the same pattern. This severe performance degradation stems from the uncoordinated access to the buffer resource which has to deal with interleaved requests coming from different sources. This fact breaks any access pattern optimization individual applications may have and any data locality that low-level schedulers aim to leverage.

The above results depict how state-of-the-art burst buffer management software such as Datawarp would schedule I/O without being aware of the cross-application interference. It is crucial for the scheduler to consider resource contention to offer applications a more efficient buffering layer. Harmonia scheduler takes into account both how storage mediums handle concurrency through $MSCA$ and how applications interfere to each other's I/O access through I_f.

IV. Design and Implementation

Harmonia is a dynamic, interference-aware, 2-way, adaptive I/O scheduler for systems equipped with shared, distributed buffering nodes. Harmonia is a middle-ware library implemented in C++. Applications can use Harmonia easily by either re-compiling their code with our library or by simply linking it by LD_PRELOAD. During the initialization of the application, Harmonia sets up all the components and establishes connection between them. All communications between application cores and Harmonia are performed via MPI one-sided operations. We developed our own distributed hashmaps, queues, and tables using MPI RMA calls (i.e., MPI_Put(*message*)) to minimize the overheads. Finally, we implemented a system profiler that can optionally run in the bootstrapping phase and collect information about the buffering system in place. This profiler helps Harmonia configure the internals according to each platform it runs.

Harmonia's design aims to mitigate the challenges mentioned in Section II. The high-level architecture of our Harmonia scheduler can be seen in Figure 4. There are several components that work together to make scheduling possible:

*1) **Harmonia Agent**:* This component represents the application. When the application starts (i.e., *MPI_Init()*), the Agent registers several information to the global Harmonia Scheduler (e.g., name and size of the job, group that owns the job etc.). The Agent acts as a "spy" for Harmonia and communicates the application's I/O behavior to the global scheduler. An I/O behavior in the context of this study is the pattern with which the application accesses the buffers. In Harmonia, instead of providing a buffer lease to an application for its entire execution time, we schedule the application's individual I/O bursts. Thus, Harmonia aims to overlap computation and I/O phases of several applications that share access to the buffers. A crucial responsibility of the Agent is to detect when an I/O phase will start and end in between of computation phases. Identifying this pattern can be very challenging. We call this task *I/O Phase Detection*. Finally, a Harmonia Agent can be distributed according to the size of the application (i.e., number of ranks). If there is an I/O forwarding layer [38], Harmonia deploys its agents on this layer. Otherwise, Harmonia reserves 1 extra core per 64 computing cores at initialization.

*2) **Harmonia Buffer Monitor**:* This component collects information from the burst buffer nodes. Specifically, the main responsibility of this component is to communicate the buffer node's status (i.e., busy - free) to the global scheduler. This is a lightweight process running in the background on buffer nodes and uses MPI one-sided operations for all communications. We use the I/O queue size (e.g., iostat) to represent how busy or not a certain buffer node is. Also, when a buffer node is flushing data to the underlying PFS, it marks itself as busy. Finally, this component keeps tracks of the capacity of each buffer node and reports back to the scheduler upon reaching a given, configurable, threshold. Optionally, Harmonia reserves a small percentage of the buffer nodes as *backup* nodes which are responsible to accept overflowing data and data ready to be flushed. The extra reserved backup nodes ensure that at any given point Harmonia can switch incoming I/O to them and flush data from the overwhelmed buffers. They also allow Harmonia to better aggregate buffered data.

*3) **Harmonia Scheduler**:* This component is a global entity that performs the actual I/O scheduling. It is multi-threaded and can be distributed. By using information collected from both the application side (i.e., Agents) and the buffering nodes side (i.e., Buffer Monitor), this component is able to effectively schedule individual I/O bursts and therefore overlap computation and I/O phases leading to better buffer utilization. Harmonia Scheduler uses several in memory structures to achieve its goals. First, it maintains an application registration table to keep track of all running applications and the phase they are in (i.e., I/O or not). Second, it maintains a buffer status table where it keeps track of how busy a node is and what is the remaining capacity on the storage devices of that node. During initialization, a system profiler populates some auxiliary structures by capturing the underlying system's metrics such as $MSCA$ and I_f and performance characteristics. Finally, as it can be seen in Figure 4, this component maintains a scheduling queue which is organized as follows (note that a *job* in the context of this study represents an I/O phase of an application): a distributed job queue for incoming I/O, a scheduled job queue, a running job list, and a completed job hashmap for statistical reasons.

At any given time, Harmonia needs to schedule a number of I/O phases into a number of available buffer nodes. To efficiently achieve this goal we propose a dynamic programming approach. Harmonia Scheduler expresses I/O scheduling as a dynamic programming optimization problem as follows:

$$OPT(n,m) = \left\{ \begin{matrix} \infty & , if\ n = 0, \\ 0 & , if\ \frac{n}{m} \geq X\ or\ m < Y, \\ \max \begin{pmatrix} C^{ij} + OPT(n^{-i}, m^{-j}), \\ OPT(n^{-i}, m), \\ OPT(n, m^{-j}), \\ OPT(n^{-i}, m^{-j}) \end{pmatrix} & , otherwise \end{matrix} \right\} \tag{3}$$

where $OPT(n,m)$ is the optimal solution of scheduling n number of I/O phases onto m number of available buffer nodes at time t, C^{ij} is the cost to schedule the i^{th} I/O phase on the j^{th} buffer with $0 \leq i < n$ and $0 \leq j < m$, X is the maximum number of concurrent applications on the same buffer (i.e., relative to I_f and $MSCA$ values), and Y is the minimum number of buffers per I/O phase (i.e., minimum parallelism degree). Our algorithm uses memoization techniques [39] to optimize the time needed to solve the scheduling problem. Therefore, Harmonia learns through time and finding the optimal scheduling decision eventually becomes very fast.

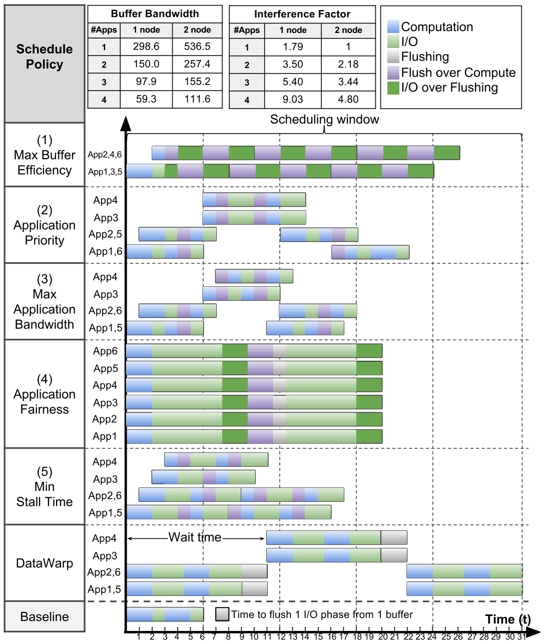
A. I/O Phase Detection

We propose three different ways to detect an I/O phase for a variety of scenarios and use cases:

*1) **User-defined**:* In this mode, the user is aware of the Harmonia scheduler and is expected to declare the start and the end of each I/O phase using the provided Harmonia pragmas (e.g., $\#pragma\ harmonia\_io\_start(...)$). The pragma defines the start of the phase, the size of the I/O within the phase, and other flags such as the priority level. When the application is compiled, the pragmas are transformed into MPI messages to the global Harmonia scheduler. This mode allows users to express their application's I/O behavior leading to better accuracy of I/O phase detection. However, not all users fully understand their I/O needs and thus, it could be challenging for the developer to clearly define an I/O phase. Additionally, some users might take advantage of this mode and overstate their I/O phases to get more buffer access time.

*2) **Source code**:* In this mode, users submit the source code along with the job. We implemented a source code analyzer where Harmonia identifies I/O phases automatically. This process takes place before the application execution. The analyzer builds a direct acyclic graph (DAG) of functions and loops which describes inter-code dependencies and helps Harmonia identify I/O calls in various blocks of code. Based on the I/O intensiveness of those blocks, Harmonia combines them together to characterize a distinct I/O phase. *I/O intensiveness* is defined as a function of the total number of files, the size of individual requests, and the count of repetitions in a loop. After identifying those I/O phases, Harmonia first injects automatically generated pragmas and then compiles the application. This mode allows Harmonia to automatically detect the I/O phases requiring less user intervention. However, in some instances, the classifier might incorrectly classify some parts of the code as I/O intensive (i.e., false-positives, -negatives). There is a trade-off between accuracy and performance that is tuned by a weighting system in place. We do not expose these weights to the user to avoid exploitation of the scheduler. Lastly, this mode is not suitable when the source code cannot be exposed due to security concerns. The accuracy, overheads, and effectiveness of our I/O intensity classifier is detailed and evaluated with real applications in [40].

*3) **Binary**(executable):* In this mode, users submit only the application binary. A normally compiled executable includes function symbols. Using these symbols (i.e., GNU *nm*), Harmonia builds the same DAG with code dependencies and proceeds to an analysis of calls (i.e., using Flint++). We then intercept all identified calls using our wrapper and we mark the start - end of an I/O phase. This results to messages destined to the global Harmonia scheduler, similarly with the *pragmas* from above. The benefits of this mode is that there is no need for user involvement or source code submission. Furthermore, the I/O phase detection is done dynamically during linking. On the downside, this method demonstrates higher chances of error in the detection when compared to the source code analyzer since it relies on information extracted from the object symbols and not the actual code. As a special case, when the binary is a striped-down executable (i.e., binary size reduction by removing debugging and other information), Harmonia proceeds to I/O phase detection by approximation. We intercept all *fopen()* and *fclose()* calls and code blocks with loops containing *fwrite()* or *fread()* to estimate the I/O phases. Clearly, this approach is the least accurate since no global view of the code can be obtained. However, this method does not require extra processing and can apply to any executable.



Buffer Bandwidth		
#Apps	1 node	2 node
1	298.6	536.5
2	150.0	257.4
3	97.9	155.2
4	59.3	111.6

Interference Factor		
#Apps	1 node	2 node
1	1.79	1
2	3.50	2.18
3	5.40	3.44
4	9.03	4.80

Policy / Metric	Average Finish Time	Buffer Utilization	Average Remaining Capacity	Buffer Efficiency	Average App Bandwidth	Time Distance Last-First Apps	Average Stall Time
1. MaxBuffEff	17	92%	100%	**96%**	36.11	18	11
2. AppPriority	14.5	55%	61%	58%	47.22	17	7.8
3. MaxAppBW	12	67%	61%	64%	**49.84**	12	6.5
4. AppFairness	20	75%	44%	59%	25.6	**0**	9
5. MinStallTime	12.5	76%	51%	64%	46.79	9	**5.8**
DataWarp	20	55%	59%	57%	33.00	22	14

Fig. 5. Harmonia scheduling policies.

Note that, misclassification of the beginning and the end of an I/O phase will not result in any catastrophic outcome in terms of performance. In scenarios where an I/O phase was inaccurately marked, applications might experience some interference, but, will successfully be scheduled and run.

B. Proposed Scheduling Policies

Harmonia schedules I/O phases on buffer nodes using a dynamic programming (DP) approach as discussed on Section IV-3. By expressing different constraints and cost functions to the DP algorithm, Harmonia is able to maximize or minimize the objective of each policy. We define a *scheduling window* as the time period in which we solve the scheduling problem. The complexity of the DP algorithm is in the order of number of I/O phases within the scheduling window. We also define a *time slice* t as the time needed to write an I/O phase on all available buffer nodes with no interference. A scheduling window has many time slices. We propose five new burst buffer I/O scheduling policies. To better demonstrate our proposed policies let us follow an example scenario shown in Figure 5 and discuss how each policy works. Our baseline application performs a computation phase that requires two time slices followed by an I/O phase and repeats this pattern twice. Therefore, if it runs unobstructed, it completes in six time slices in total. Assume we have 6 instances of this application coming into the system and 2 available buffers. Also assume that the buffers can fit up to 4 I/O phases before they exceed their capacity. Finally, as a reference, we measured the I/O bandwidth and the interference factor of our hypothetical system for all combinations of four applications and two buffers (e.g., 1 app on 1 buffer achieves 298MB/s and has a slowdown of 1.79x compared to the baseline, etc.). Results shown at the top of Figure 5 are real measurements of a system equipped with two SSD-based burst buffers.

DataWarp schedules applications through the central batch scheduler (e.g., Slurm) based on the remaining capacity of the buffers. Each application asks for a reservation of predefined buffer capacity. Once the resources are available, the application acquires the reservation and it can freely write data to BBs. Otherwise, the application ("app" for the rest of the paper) will remain in the batch scheduler's queue until BB capacity becomes available. In our example, DataWarp will pick apps 1 and 2 based on FCFS order and will deploy two PFSs on the 2 available BBs (i.e., one for each application). This will lead to collocation of the apps on the buffers and therefore increased interference. The rest of the apps will be waiting for the buffers to become available. Upon completion of the first two apps, DataWarp will flush the contents of the buffers to the remote PFS and proceed with the next two apps. This process will be repeated for the last two remaining apps.

*1) **Harmonia - Maximum Buffer System Efficiency**:* In this policy, Harmonia aims to optimize the buffering system efficiency by maximizing the buffer utilization (i.e. minimizing idleness) while at the same time maximizing the average available capacity of the disks. To maintain high available remaining capacity, Harmonia hides flushing behind computation phases of the same or another running application, and thus, keeps the buffers busy at all times. This policy is ideal when most of the apps share similar characteristics and no one app requires special treatment. The cost function in the DP algorithm (i.e., C_{ij} from eq. 3) is defined as:

$$Cost(i,j)_{Buffer-Efficiency} = \frac{\overline{B_U} + \overline{C_R}}{2} \tag{4}$$

where $\overline{B_U}$ is the average buffer utilization and is calculated as the ratio of the time when buffers are serving an app over the completion time of the last app, $\overline{C_R}$ is the average remaining capacity of the buffers. Harmonia considers if scheduling the i_{th} I/O phase on the j_{th} buffer will increase the buffer utilization while maintaining the same or smaller average remaining capacity leading to better buffer efficiency. In our example, Harmonia, under this policy will schedule app 1 at t=0 and will run on top of both buffers hence maximum I/O bandwidth. App 2 will start at t=2, effectively overlapping its computation phases with the I/O phase of app 1. Additionally, the buffers will start flushing as soon as data are available, practically at t=3, and it will keep flushing during both computations and I/O phases. The rest of the apps will be scheduled once one of the previous apps finishes. This ensures the constraints of our buffering system are intact. The number of concurrent accesses on one buffer are kept low (one I/O phase and flushing) and thus, the interference factor remains low.

*2) **Harmonia - Application Priority**:* In this policy, Harmonia aims to provide system administrators the infrastructure to prioritize the buffer access of specific applications. Each I/O phase is flagged with a priority either given by the user or automatically extracted by Harmonia's I/O phase detection. Under this policy, Harmonia will first sort all I/O phases according to the priority and then execute them exclusively on the buffers guaranteeing maximum bandwidth and zero interference. For I/O phases with same priority, Harmonia offers two modes: equally share the available buffering resources (default mode) or execute them in first-come-first-serve (FCFS) fashion. The first mode maintains the defined priority whereas the second respects fairness. If an I/O phase with higher priority comes to the system, it will be scheduled before earlier submitted phases. Note that Harmonia will not interrupt a running I/O phase to place a higher priority one. Instead, it will re-schedule all remaining phases along with the new incoming ones by enforcing the new priority order. In our example, let us assume that after sorting applications based on priority, we have the following pairs (app-priority): app1-p5, app2-p4, app3&4-p3, app5-p2, and app6-p1. This dictates exactly how each I/O phase should be scheduled. Note that for app 3 and 4 that have the same priority 3, Harmonia schedules them to start at t=6 by sharing the two available buffers and after ensuring that the last I/O phase of the app with higher priority is completed and properly flushed (i.e., at t=7 flushing happens overlapped with computation). Whenever possible, overlapping of computation with I/O or flushing helps the overall average completion time without violating the priorities.

*3) **Harmonia - Maximum Application Bandwidth**:* In this policy, Harmonia aims to maximize the bandwidth each application gets by the buffering layer. It can achieve this by scheduling I/O phases exclusively. In other words, under this policy, Harmonia tries to offer higher parallelism while minimizing the interference factor I_f. The cost function in the DP algorithm is defined as:

$$Cost(i,j)_{MaxAppBW} = \frac{\sum_1^n \frac{IOSize}{CompletionTime}}{n} \tag{5}$$

where n is the number of I/O phases to be scheduled. Effectively, this policy looks on how much I/O the app did at its completion time. This policy might schedule an I/O phase later if that ensures maximum buffer bandwidth. The overall goal is to equally offer all scheduled applications maximum bandwidth. The selection of which application goes first is performed randomly. This policy is ideal for crucially sensitive applications to latency and critical mission applications. In our example, each app is scheduled to have access to both buffers maximizing the bandwidth during an I/O phase. Harmonia carefully overlaps computation phases with I/O phases or flushing while maintaining the constraint of the policy. To be able to offer maximum bandwidth, we can see that up to 2 concurrent applications are scheduled. For instance, at t=3, app 1 and app 2 are running with only app 1 being in an I/O phase. Under this policy, no I/O phase will tolerate interference with another I/O or flushing in contrast with the first policy.

*4) **Harmonia - Application Fairness**:* In this policy, Harmonia aims to offer absolute fairness in terms of waiting and execution time to all scheduled applications. Resources are divided equally into all applications. Harmonia ensures that it does not violate the constraints though. In our study in Section III-B we found that more than three applications on one buffer node is really hurtful in terms of performance (i.e., I_f greater than 5x). This policy takes into account this fact when dividing the resources. If the total number of I/O phases to be scheduled cannot fit into the available buffers, Harmonia randomly selects which ones to schedule at this point in time and which ones later. The randomness ensures that no application will be favored over time. The cost function in the DP algorithm is defined as:

$$Cost(i,j)_{Fairness} = C_{time}LastApp - C_{time}FirstApp \tag{6}$$

where C_{time} is the completion time of the application. Completion time encapsulates both the wait time to be scheduled and the bandwidth available to each application since it is a function of both. This cost represents the time distance between the first and the last application to finish. Typically, a totally fair system will have this distance equal to zero. All applications are expected to have the same completion time. This policy is effective in homogeneous systems and also to applications that can sacrifice high performance in favor of earlier access to the buffers. In our example, Harmonia first checks if the 6 applications violate the constraint of up to 3 apps to 1 buffer. Coincidentally in this case, Harmonia can indeed start all 6 applications at the same time t=0 by scheduling apps 1-3 on the first buffer and apps 4-6 on the second buffer node. Another constraint is that the buffers can fit up to 4 I/O phases before they run out of space. Harmonia will choose to tolerate added interference and will start a background flushing (notice that between t=7 and t=9 the I/O overlaps with flushing which continues during the next computation). The decisions made by Harmonia result in a situation where all 6 applications are allowed to continue execution even though they get less bandwidth and thus, extended I/O phases due to interference.

*5) **Harmonia - Minimum Application Stall Time**:* In this policy, Harmonia aims to minimize the average time applications spend in waiting to be scheduled or prolonged I/O phases due to interference. The cost function in the DP algorithm is defined as:

$$Cost(i,j)_{StallTime} = \frac{\sum_1^n (W_{time} + I_{time})}{n} \tag{7}$$

where W_{time} is the time waiting to be scheduled and I_{time} is the extra time that the I/O phase will need to finish due to interfering with one or more other I/O phases. Harmonia will try to balance performance and stall time by sacrificing some bandwidth. However, since this policy takes into account all different types of stalling including the I_{time}, Harmonia needs to find the best possible way to collocate apps while offering a reasonable performance. The goal of this policy is to offer minimum stall time on average. The selection of which app starts first is random ensuring that in a period of time all apps will experience similar stall time. In our example, Harmonia will schedule app 1 on the first buffer at t=0, app 2 on the other buffer at t=1, and app 3 on the first buffer overlapping computation and I/O with app 1. Each individual buffer, at any time slice t will either execute an I/O phase or will be flushing its contents while overlapping with computations.

Additional discussion: Throughout this section we described Harmonia's scheduling policies using a motivating example with write I/O phases. However, our policies work great for read operations as well. Instead of a write I/O phase followed by a flushing, read operations involve a prefetching (i.e., loading of the data) before the read I/O phase. The benefits of each policy remain the same. Each type of I/O (e.g., read or write) is identified through the I/O phase detection and therefore Harmonia is able to pro-actively fetch the required data. Furthermore, for workloads that require reading after writing, Harmonia employs a hinting system where an I/O phase is marked as ”cached” or ”flushable”. Using the hints, Harmonia will not schedule other I/O phases on the same buffer until it is safe to do so allowing an application to cache data for future read operations. Lastly, for asynchronous I/O calls, Harmonia can utilize the traffic service classes implemented in InfiniBand networks (i.e., traffic class field *TClass* in Mellanox) to handle both I/O and compute traffic. The *pragmas* that describe the start-end of an I/O phase are inserted normally, as if the calls were synchronous. Harmonia's 2-way design allows the scheduler to track when the actual I/O happens by utilizing information from the Buffer Monitor component.

V. Experimental Results

A. Methodology

Testbed: All experiments were conducted on Chameleon systems [36]. More specifically, we used the bare metal configuration offered by Chameleon. The total experimental cluster consists of 1024 client MPI ranks (i.e., 32 nodes), 4 buffer nodes, and 8 PFS servers. Each node has a dual Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz (i.e., a total of 48 cores per node), 128 GB RAM, 10Gbit Ethernet, and a local HDD for the OS. Each burst buffer node has the same internal components but, instead of an HDD, it has a 200GB SSD. The cluster OS is CentOS 7.1, the PFS we used is OrangeFS 2.9.6.

Software: First, we wrote our own synthetic benchmark where we can easily execute computation and I/O phases interchangeably. We emulated computation phases by artificial math kernels. The configuration of our tests for Harmonia's evaluation is: 8 instances of the benchmark, each instance runs with 128 processes for a total of 1024 MPI ranks, each instance writes 192GB of data in 12 I/O phases (i.e., 16GB per phase), our four buffer nodes have 800GB aggregate capacity and thus can fit 48 I/O phases before they need to flush data to PFS. We also use two real science applications: Vector Particle-In-Cell (VPIC), a general purpose simulation code for modeling kinetic plasmas, and Hardware Accelerated Cosmology Code (HACC), a cosmological simulation. We run the simulations with 1536 MPI ranks with each rank performing 32MB of I/O. Both of these simulations perform I/O periodically in time steps (i.e., iterations). We run 16 time steps for both simulations resulting to total I/O of 768GB.

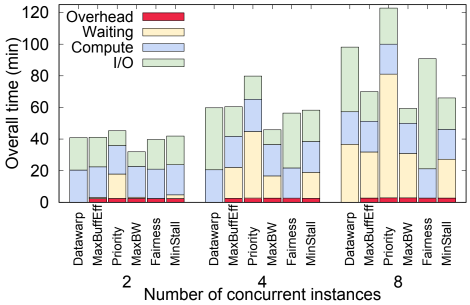


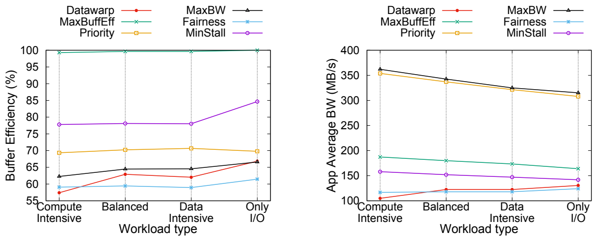
Fig. 6. Execution time and overheads.

Workloads: We use four input workloads a) Compute-Intensive: computation phases run substantially longer in time than IO, b) Balanced: where the computation and IO phases are approximately equal in time, c) Data-Intensive: I/O phases run substantially longer in time than computation, d) Only-I/O: where there is no computation between I/O. We use four distinct metrics which can describe a wide variety of desired goals: maximum buffer efficiency (MaxBuffEff), maximum bandwidth (MaxBW), fairness, minimum stall time (MinStall). The calculation of the metrics stems from equations (4)-(7). We measure each metric for each Harmonia policy and we compare the values with DataWarp. We also use the overall execution time in minutes to compare timings of individual operations such as I/O, waiting, completion time, etc.

B. Library Performance and Overheads

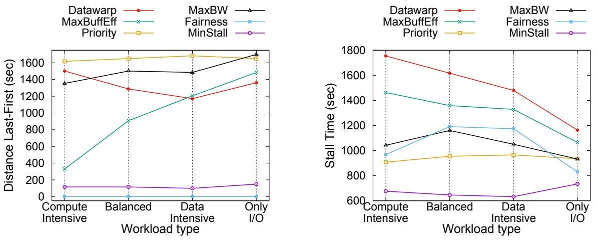
Figure 6 presents the breakdown of the overall average execution time of scheduling several instances of our benchmark. Overall average time includes time waiting to be scheduled, computation time, and time to perform I/O. Overheads include time to complete the I/O phase detection, which is executed offline but still needs to be measured, and time to optimally solve the scheduling problem. We scale the number of concurrent instances (i.e., arriving to the scheduler) from 2 to 8 and we compare Harmonia's policies to DataWarp. Note that the buffers can fit up to 4 instances before they need to flush data to PFS. Lastly, Figure 6's goal is to compare Harmonia's policies against DataWarp and not to each other since every one was designed to cover different workloads and scenarios.

As it can been seen, DataWarp does not impose any overheads since it gives access to the buffers by a mounting point. However, DataWarp does not account for interference and will schedule the instances on all available buffers. Thus, for 2 and 4 concurrent instances of the benchmark, DataWarp demonstrates increased I/O time due to interference. Additionally, if there is no buffer capacity left, DataWarp will queue the remaining applications. This can be seen in the 8 concurrent instances cases where DataWarp schedules the first 4 instances and the remaining are left to the queue resulting in an overall average execution time of around 6000 sec. In



(a) Max Buffer Efficiency (b) Max Bandwidth

Fig. 7. Scheduling metrics: MaxBuffEff - MaxBW.



(a) Fairness (b) Stall Time

Fig. 8. Scheduling metrics: Fairness - Stall Time.

contrast, Harmonia's policies schedule I/O phases and take advantage of any computation time to either overlap the I/O or flushing the buffers. The overhead is the same between policies and is about 4% of the overall execution time on average.

Harmonia's MaxBuffEff offers competitive performance with DataWarp for 2 and 4 concurrent instances. However, it performs 30% faster for 8 instances since it manages to utilize the buffers during computations. This figure reports the average time for all applications and not of the prioritized one, however, we include Harmonia's Priority policy's results for completeness. Harmonia's MaxBW policy demonstrates the best overall execution time for all scales. It consistently offers the shortest I/O time since it avoids the negative effects of interference and outperforms DataWarp by 40% for the 8 instances. Harmonia's Fairness offers zero waiting time on the expense of slower I/O time since it schedules instances allowing interference. The slower I/O time is especially apparent on the 8 concurrent instances cases where it demonstrates 4100 seconds I/O time. This policy still manages to outperform DataWarp by 10% since it avoids queuing and shows that if the scheduler is aware of the interference in the buffers, it can offer a scheduling scheme that can result in decent average performance to all running applications. Harmonia's MinStall policy offers a competitive performance by balancing waiting time and interference and outperforms DataWarp by 30%.

C. Scheduling Metrics

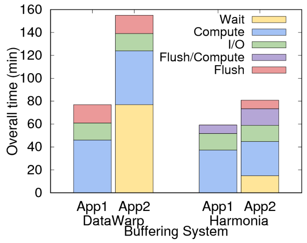
Harmonia is a burst buffer scheduler and as such we compare its performance with scheduling metrics. We present the results of scheduling 8 instances of our benchmark and we compare Harmonia's scheduling policies with DataWarp. We used various types of input to show how the application characteristics affect the scheduling performance.

Max Buffer Efficiency: As it can been seen in Figure 7 (a), Harmonia's MaxBuffEff policy exhibits very high buffer efficiency for all inputs by keeping the buffering nodes always busy. When compared with DataWarp's efficiency score of 55% for the compute-intensive input, Harmonia is almost **2x** more efficient. Harmonia's Fairness is the least efficient among all Harmonia's policies and scored around 60% regardless of the input. This metric is important in terms of system utilization and power efficiency. By over-provisioning the buffering resources and by maintaining high buffer utilization, Harmonia can lead to more efficient systems.

Max Bandwidth: As it can been seen in Figure 7 (b), Harmonia's MaxBW and Priority policies sustain the highest average bandwidth. This figure presents the average bandwidth each application experiences on average. If there would be only one instance of the benchmark, then all solutions would offer the same bandwidth out of all buffering nodes. However, in this test, we schedule 8 instances of the benchmark and thus, DataWarp can only get roughly 115MB/s across all inputs due to interference in the data accesses and due to non-overlapping compute-I/O phases. Harmonia, on the other hand, managed to achieve an average bandwidth of around 350MB/s effectively outperforming DataWarp by **3x**. This metric is important in terms of pure I/O performance. Higher achieved bandwidth leads to lower execution time and therefore more chances to schedule more applications within a certain period of time. Harmonia can lead to more capable I/O buffering platforms.

Fairness: As it can been seen in Figure 8 (a), Harmonia's Fairness policy works as intended and performs the best for all input workloads. DataWarp suffers in this metric since it needs to serialize the applications in a FCFS fashion (i.e., based on the available buffer capacity inside the reservation), and thus, imposes a certain order violating the fairness metric. Lower scores in this metric means that the system schedules incoming applications in a fairer way. DataWarp scores more than 1500 for compute-intensive input and around 1200 for data-intensive. Harmonia outperforms DataWarp by more than **10x**. This metric reflects the behavior of the scheduler and does not translate to higher performance or buffer efficiency. It does, however, provide higher flexibility to system administrators offering fair I/O buffering services across teams and projects.

Min Stall Time: As it can been seen in Figure 8 (b), under Harmonia's MinStall policy, applications should expect an average stall time of around 620 seconds for all workloads tested, whereas the same metric in DataWarp has a value of more than 1700 seconds for compute-intensive and 1200 for Only-I/O. DataWarp offers exclusive access to the buffers and thus, it does not impose any extra stall time between each I/O phase but only between different applications. On the other hand, Harmonia schedules I/O phases independently and therefore, the scheduler might block one application's access to the buffers in order to finish another application's I/O phase. However, Harmonia outperforms DataWarp by more than **3x** and offers applications quick access to the buffers.



Description	DataWarp	Harmonia
Min Completion Time	76.9	59.2
Max Completion Time	155.1	80.8
Average Completion Time	116	70
Average Wait Time	38.45	7.45

Test Metrics in Minutes

Fig. 9. VPIC: Draining Effect in Scheduling

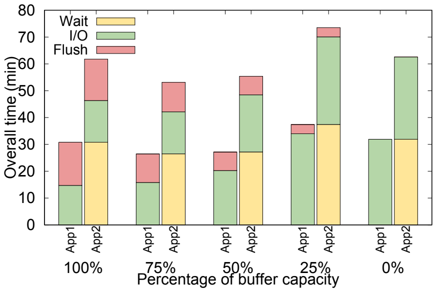


Fig. 10. HACC-IO: Buffers at full capacity.

D. The Buffer Draining Effect

The remaining capacity of the buffers is very crucial to scheduling. In DataWarp, applications reserve burst buffers based on capacity. When the buffers are full, data need to be flushed to PFS for persistency. We call this process buffer *draining*. To evaluate the effect buffer draining has on scheduling we schedule two instances of VPIC (i.e., each with 16 time steps). Only one instance can fit in the buffers and therefore the buffers need to flush first before scheduling the next instance. In each step, VPIC performs computations and then writes data to the buffers. Harmonia, leverages these computation phases to drain the buffers and therefore it is able to schedule both instances and hide the flushing between each time step. Figure 9 demonstrates the results. As it can be seen, Harmonia outperforms DataWarp by **2x**. The average completion time is reduced from 116 to 70 min in Harmonia.

One of the configurable parameters in Harmonia is the flushing threshold. Harmonia will drain the buffers only when they reach the remaining capacity threshold. In Figure 10 we show the results of running HACC-IO, a pure I/O kernel with no computation phases to overlap draining (i.e., a particular difficult case for the buffers). We change the above threshold from 100 to 0%. Since only one instance of the application can fit in the total capacity of the buffers, the second instance has to wait. In the 100% case, Harmonia behaves exactly as DataWarp. It uses all 4 buffer nodes and gets the maximum bandwidth. It flushes at the end before it schedules the next application. In the 0% case, the buffers accept incoming I/O while they are draining at the same time. This means that interference is present and the I/O performance is affected. In contrast, Harmonia can utilize its special *backup* flusher nodes. This is a trade-off between the rate of the incoming I/O and the buffer draining. In the 75%, when the threshold is reached, one buffer node becomes the flusher and the incoming I/O is directed to the remaining three with a penalty in performance. However, as it can be seen, the overall performance when compared to DataWarp (i.e., the 100% case) is higher by almost 20%. Similarly, in the 50% case, we see that I/O performance is getting significantly worse due to the concurrent draining. In summary, Harmonia, by grouping the available burst buffer nodes, offers higher performance than DataWarp.

VI. Related Work

IME [41], a burst buffer platform, focuses on eliminating I/O wait times and increasing resource utilization. However, it does not take I/O interference within devices into consideration. Our study shows that the interferences largely vary between different kinds of devices. Harmonia mitigates these issues by implementing several interference-aware scheduling policies optimizing global I/O performance of the system.

Traditional I/O schedulers for PFS such as ADIOS [42], a client level I/O scheduler which provides a simple API to handle I/O of application, allow applications to create an I/O pipeline by using staging nodes. However, ADIOS does not consider multiple applications running in the system. TRIO [43], is a burst buffer orchestration framework, to efficiently move the large check-pointing dataset to PFS. This study focuses on PFS contention during flushing but does not apply for contentions of BB during incoming I/O. In [44], the authors identified and quantified the negative effects of interference in the context of burst buffers and proposed via simulations a scheduling technique to mitigate such effects. Finally, [45] proposes a model to reduce I/O congestion and defines few metrics to optimize the overall system using a global I/O scheduler. This takes into account within application congestion, but it does not include cross-application interference caused due to resource contention.

How SSDs are affected when used as scratch space for applications has been explored in [46], [47], [48]. Multiple write and delete operations trigger SSD's garbage collection which degrades the performance greatly. They introduce disk level scheduling to perform preemptive garbage collection for better overall performance. However, they optimize I/O on a device level which does not have the global view of the system. Harmonia utilizes information from both the application and buffer node level leading to better I/O scheduling.

VII. Conclusion

In this study, we present the design and implementation of Harmonia, a new dynamic I/O scheduler that is interference-aware, operates in a finer granularity, and is adaptive to the current system status. Harmonia's several scheduling policies make it capable to handle a wide variety of workloads and different scenarios giving system administrators many tools to achieve their respective goals. By overlapping computation and I/O phases, calculating I/O interference into its decision making process, and by employing a novel dynamic programming algorithm, Harmonia outperforms state-of-the-art burst buffer systems by 3x leading to better resource utilization.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grants no. CCF-1744317, CNS-1526887, and CNS-0751200.

References

[1] B. Dong, X. Li, L. Xiao, and L. Ruan, "A new file-specific stripe size selection method for highly concurrent data access," in *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*. IEEE, 2012, pp. 22–30.

[2] A. Shoshani and D. Rotem, *Scientific Data Management: Challenges, Technology, and Deployment*. CRC Press, 2009.

[3] P. J. Braam, "The Lustre storage architecture," *Cluster File Systems, Inc*, 2004. [Online]. Available: https://bit.ly/2uYuQzj

[4] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, vol. 2. Monterey, CA: Usenix, 2002, pp. 231–244.

[5] R. B. Ross, R. Thakur *et al.*, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th annual Linux Showcase and Conference*, 2000.

[6] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.

[7] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[8] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 1–25.

[9] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 217–228, 2009.

[10] S. Kannan, A. Gavrilovska, K. Schwan, D. Milojicic, and V. Talwar, "Using active nvram for i/o staging," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*. ACM, 2011, pp. 15–22.

[11] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 219–230.

[12] N. Mi, A. Riska, Q. Zhang, E. Smirni, and E. Riedel, "Efficient management of idleness in storage systems," *ACM Transactions on Storage (TOS)*, vol. 5, no. 2, p. 4, 2009.

[13] Y. Kim, R. Gunasekaran, G. M. Shipman, D. Dillow, Z. Zhang, B. W. Settlemyer *et al.*, "Workload characterization of a leadership class storage cluster," in *Petascale Data Storage Workshop (PDSW), 2010 5th*. IEEE, 2010, pp. 1–5.

[14] Lawrence Livermore National Lab, "Large Memory Appliance/Burst Buffers Use Case." [Online]. Available: http://bit.ly/2Lv4yiG

[15] CRAY Inc, "Datawarp technology," 2017. [Online]. Available: http://bit.ly/2NOS3uO

[16] Whitt, Justin L, "Oak Ridge Leadership Computing Facility: Summit and Beyond," 2017. [Online]. Available: http://bit.ly/2Opd28M

[17] International Data Corp (IDC), "IDC Report: SSD price premium over disk is falling." [Online]. Available: http://bit.ly/2LWJdKS

[18] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[19] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*, 2014.

[20] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing Output Bottlenecks in a Supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 8.

[21] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the io performance of petascale storage systems," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–12.

[22] Y. Hashimoto and K. Aida, "Evaluation of Performance Degradation in HPC Applications with VM Consolidation," in *Networking and Computing (ICNC), 2012 Third International Conference on*. IEEE, 2012, pp. 273–277.

[23] J. Lofstead and R. Ross, "Insights for Exascale IO APIs from Building a Petascale IO API," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12.

[24] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, 2014.

[25] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent i/o interference," in *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, 2016, pp. 371–380.

[26] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications Under Congestion," Ph.D. dissertation, LIP, 2014.

[27] S. Thapaliya, A. Moody, and K. Mohror, "Inter-application coordination for reducing i/o interference," 2010.

[28] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[29] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[30] Oak Ridge National Lab, "Leadership Computing Requirements for Computational Science." [Online]. Available: https://bit.ly/2Oqa03T

[31] A. Kougkas, H. Devarajan, and X.-H. Sun, "IRIS: I/O Redirection via Integrated Storage," in *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS)*. ACM, 2018.

[32] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–11.

[33] D. Brown, P. Messina, D. Keyes, J. Morrison, R. Lucas, J. Shalf, P. Beckman, R. Brightwell, A. Geist, J. Vetter *et al.*, "Scientific grand challenges: Crosscutting technologies for computing at the exascale," *Office of Science, US Department of Energy, February*, pp. 2–4, 2010.

[34] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. V. Straalen, D. Trebotich, C. Tull, G. Weber, N. J. Wright, and K. Antypas, "Accelerating science with the nersc burst buffer early user program," 2016.

[35] NERSC, "Burst buffers known issues." [Online]. Available: https://bit.ly/2LM53UH

[36] Chameleon.org, "Chameleon system," 2017. [Online]. Available: http://bit.ly/2uYbKcJ

[37] D. Skinner and W. Kramer, "Understanding the causes of performance variability in hpc workloads," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 2005, pp. 137–149.

[38] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2009.

[39] J. Jaffar, A. E. Santosa, and R. Voicu, "Efficient memoization for dynamic programming with ad-hoc constraints." in *AAAI*, vol. 8, 2008, pp. 297–303.

[40] Hariharan Devarajan, Anthony Kougkas, Prajwal Challa, Xian-He Sun, "Vidya: Performing Code-Block I/O Characterization for Data Access Optimization," Tech. Rep., 2018.

[41] M. Romanus, R. B. Ross, and M. Parashar, "Challenges and considerations for utilizing burst buffers in high-performance computing," *arXiv preprint arXiv:1509.05492*, 2015.

[42] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello adios: the

challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[43] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "Trio: burst buffer based i/o orchestration," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*. IEEE, 2015, pp. 194–203.

[44] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody, "Managing i/o interference in a shared burst buffer system," in *Parallel Processing (ICPP), 2016 45th International Conference on*. IEEE, 2016, pp. 416–425.

[45] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the i/o of hpc applications under congestion," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 1013–1022.

[46] J. Lee, Y. Kim, G. M. Shipman, S. Oral, and J. Kim, "Preemptible i/o scheduling of garbage collection for solid state drives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 247–260, 2013.

[47] J. Lee, Y. Kim, J. Kim, and G. M. Shipman, "Synchronous i/o scheduling of independent write caches for an array of ssds," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 79–82, 2015.

[48] B. Jun and D. Shin, "Workload-aware budget compensation scheduling for nvme solid state drives," in *Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE*. IEEE, 2015, pp. 1–6.