Design and Implementation of Dynamic I/O Control Scheme for Large Scale Distributed File Systems

Sunggon Kim[a,*], Alex Sim[b], Kesheng Wu[b], Suren Byna[b] and Yongseok Son[c]

[a]Seoul National University of Science and Technology
[b]Lawrence Berkeley National Laboratory
[c]Chung-Ang university

ARTICLE INFO

Keywords:
High-performance computing; Distributed dynamic resource management; Autonomous control;Parallel and Distributed File System; Cloud System

ABSTRACT

In this work, we have analyzed the input/output (I/O) activities of Cori, which is a high-performance computing (HPC) system at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory. Our analysis results indicate that most users do not adjust storage configurations but rather use the default settings. In addition, owing to the interference from many applications running simultaneously, the performance varies based on the system status. To configure file systems autonomously in complex environments, we developed DCA-IO, a dynamic distributed file system configuration adjustment algorithm that utilizes the system log information to adjust storage configurations automatically. Our scheme aims to improve the application performance and avoid interference from other applications without user intervention. Moreover, DCA-IO uses the existing system logs and does not require code modifications, an additional library, or user intervention. To demonstrate the effectiveness of DCA-IO, we performed experiments using I/O kernels of real applications in both an isolated small-sized Lustre environment and Cori. Our experimental results shows that our scheme can improve the performance of HPC applications by up to 263% with the default Lustre configuration.

1. Introduction

High-performance computing (HPC) is being widely adopted owing to the increasing demand for large-scale computation and big data [28, 36, 23]. HPC applications have many different characteristics compared to traditional applications because they utilize a large amount of computational power. Moreover, HPC applications often produce a significantly greater volume of data than traditional applications do [4, 19, 29]. Therefore, many HPC applications perform checkpointing, which stores intermediate data to protect them from unexpected power outages or scheduling. Because applications wait for the completion of I/O before performing further computations, the performance of the application is strongly related to the I/O performance. Thus, it is becoming increasingly important to enhance the I/O performance to improve the overall utilization of HPC systems.

Because the HPC environment storage architecture is inherently different from traditional architecture, careful considerations must be made to efficiently exploit the I/O performance. For example, instead of local file systems, such as EXT4 [25] and XFS [35], parallel and distributed file systems, such as Lustre [30] and Ceph [38], are widely used in many HPC environments to achieve high performance, reliability, and scalability. Many systems provide various configuration options to allow users to specify the number of nodes to place data (stripe count) and the size of the data chunk to be placed in each node (stripe size) to parallelly utilize multiple nodes of distributed file systems. To fully exploit the I/O performance of parallel and distributed file systems, it is crucial to analyze the I/O behavior of the application and adjust the configurations accordingly.

In addition, the parallel and distributed file system is shared by many applications, and its performance is affected by interference from other applications. As many users simultaneously access the file system, it is crucial to provide stable performance when multiple applications simultaneously perform I/O operations. Thus, many parallel and distributed file systems allow users to control which storage node is to be utilized (starting offset). To fully exploit the file system, it is important to consider the storage nodes accessed by other applications and adjust the configurations to avoid I/O contention on the nodes.

In previous studies, researchers have attempted to improve the I/O performance of applications by understanding their I/O behaviors and adjusting the distributed file system configurations. Yu et al. [40] characterized the I/O patterns from the applications and proposed optimal Lustre configurations depending on the characteristics determined from the experiment results. You et al. [39] proposed an auto-tuning framework that models the application and runs the model in a separate system with multiple configurations to determine the optimal configuration. Lofstead et al. [20] measured the extent of interference caused by multiple simultaneous applications and designed an adaptive algorithm that balances the I/O workloads generated from HPC applications. Dorier et al. [11] categorized strategies to avoid interference and developed a framework that alleviates I/O interference by dynamically selecting appropriate policies. Our study is in line with these studies in finding the optimal configuration and minimizing interference by adjusting the configurations.

In this article, we first present the result of analyzing

sunggonkim@seoultech.ac.kr (S. Kim); asim@lbl.gov (A. Sim); kwu@lbl.gov (K. Wu); sbyna@lbl.gov (S. Byna); sysganda@cau.ac.kr (Y. Son)
ORCID(s):

the I/O activities in Cori, which is an HPC environment, at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkley National Laboratory. Although many previous investigations [4, 15] have reported that the use of the optimal configuration can significantly improve the application performance, the result of our analysis verifies that a vast majority of users use the default configuration. In our previous work [17], we focused on the analysis of file system configurations and their effects on the application performance. This article further investigates the performance variation when an identical configuration is used. Based on the analysis, in a distributed system, limited storage resource is shared by multiple applications. Thus, it is important to consider the effects of interference between applications, and careful considerations are required to fully exploit the file system.

To improve the I/O performance of applications and overall storage utilization, we developed DCA-IO, an algorithm that dynamically configures the Lustre file system. When a new application is submitted and no information on I/O behavior on the submitted application is available, DCA-IO uses statistical analyses of other applications that previously ran on the HPC system to minimize the modeling and training overhead. By analyzing the history of applications in the same environment, DCA-IO can adjust the configuration without knowing the specific I/O behavior of the submitted application. After the application is executed and the information is available, DCA-IO utilizes the information from the previous executions and optimizes the configurations using a set of rules. Finally, DCA-IO continues to improve the distributed file system configurations dynamically as the application recurs multiple times. In addition to the application-specific configuration schemes from the state-of-art scheme [27], DCA-IO adjusts the configuration to minimize interference between multiple applications. This can not only reduce interference but also improve the overall I/O performance and efficiency in the large distributed system where multiple applications with diverse I/O characteristics are executed. Our experimental results with real HPC applications demonstrate that the use of the proposed algorithm can lead to improvements in the I/O performance of the applications by up to 75% in an isolated environment and 50% in Cori, and in the case of simultaneous execution, the performance can be improved by up to 263% compared with the default configuration.

The remainder of this article is organized as follows: Section 2 describes the background and motivation of the study, and Section 3 discusses the analysis results. Section 4 presents the design and implementation of DCA-IO, and Section 5 shows the experimental results. Section 6 discusses related works, and Section 7 concludes the article.

Thank you for your valuable comment. We agree that the application name can be inconclusive to determine the I/O behavior of an application. However, inspecting the input data or the code can induce a large overhead. Since our goal is to design a lightweight I/O control scheme, we used the application name. To accommodate the comment, we added an explanation of why we used the application name in the design section.

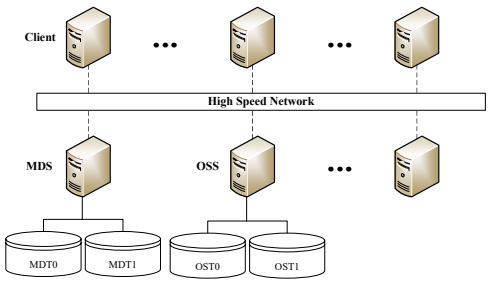


Figure 1: Architecture of Lustre file system.

2. Background

2.1. Lustre File System

Lustre file system [30] is a parallel and distributed file system used in many HPC environments including Cori. Figure 1 illustrates the overall architecture of Lustre file system, which consists of two main servers.

- Metadata server (MDS) stores and provides the metadata of the file system such as the file names, permission information, and directories. Each MDS consists of one or more metadata targets (MDTs) which are disks used to store actual data.
- Object storage server (OSS) stores file data on one or more object storage targets (OSTs). The maximum throughput and maximum capacity of OSS are calcuated using the sum of maximum throughput and maximum capacity of each OST, respectively.

When a client creates and writes a new file, the file can be distributed over multiple OSSs with differently sized file chunks, which can be configured using stripeCount and stripeSize parameters. These configurations are only affecting file layout of the specific directory and does not affect other directory utilized by other applications. By adjusting stripeCount, the client can improve the parallelism because multiple OSSs can be used in parallel. By adjusting stripeSize, the data from a particular process can be stored in a contiguous space. The performance of applications can be improved by several orders of magnitude with ideal stripeCount and stripeSize [39].

The files created by multiple clients are distributed over OSSs. OSSs for storing data can be selected by configuring startingOffset. By adjusting startingOffset, Lustre file system writes the data consecutively from the starting OSS. In the HPC environment, configuring startingOffset can mitigate the interference caused by multiple applications because assigning different sets of OSSs can isolate the performance of the applications.

By default, Lustre file system selects OSSs and allocates data objects to OSSs using two algorithms [21].

- Round-robin allocator evenly distributes the data among OSSs when they have similar amounts of free space.

- Weighted allocator changes the OSS order by checking the available capacities of OSSs.

The allocation methods are selected alternatively to balance the capacities of OSSs. This is done to load balance multiple OSSs in the file system. However, these algorithms do not necessarily deliver the best performance in all cases because the applications have different configurations and the algorithms are optimized for capacity management. To improve the performance of the complex HPC file system, it is important to analyze the system and understand the characteristics of the applications.

2.2. Analyzing and Optimizing I/O Performance in HPC Environment

To analyze the application behavior, many previous studies have proposed system wide tools for understanding application behavior in the HPC environment [9, 31]. Regarding I/O, Darshan I/O characterization tool, developed by Argonne National Laboratory is widely used in many HPC environments [8]. Darshan is widely used in complex HPC systems as it is scalable to thousands of cores. In addition, it is designed with full time deployment in mind as it is light weight. Thus, Cori collects darshan logs for all the applications by default which allows system administrators to detect anomaly in I/O performance after critical events such as software and hardware upgrade.

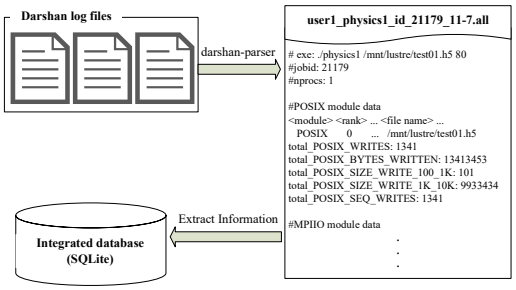
When the application is compiled, Darshan inserts codes that intercept *MPI_Init()* to initialize Darshan data structures and *MPI_Finalize()* to terminate the Darshan process. When the application runs, Darshan captures I/O related function calls from the HPC applications on a per-file and per-process basis in a light-weight manner. After the application terminates, it aggregates the collected information and writes it in a file format. Because Darshan has negligible overhead and captures the complete record of the I/O function calls, it has been used in many previous studies to understand I/O behavior and create an application-specific model [8, 34]. For example, Patel et al [29] used darshan logs of large scale system and analyzed access pattern in terms of file usage. While there are other system resources such as network and memory usage, Cori currently does not support them as default as they are complex to monitor and induce large overhead. Thus, in this paper, we focus on the I/O performance in the system.

3. Analysis

In this section, we explain the methodology used to collect information from the existing Darshan logs of Cori and present the analysis results.

3.1. Collecting Darshan Logs

To determine the I/O activities of the HPC applications and Lustre file system configuration used by the user, we have analyzed Darshan logs from Cori over two months (October to November 2017). In Cori, Darshan is configured as the default I/O characterization tool, and Darshan logs are stored automatically after each application execution [33]. Because the logs are stored in a raw file format, logs must undergo a few transformations, as illustrated in Figure 2.



progName	userID	jobID	numProc	numOST	stripeSize	seqWrite (%)	requestMore1M (%)
physics1	user1	21179	1	4	1048576	100	74.3
physics1	user5	21230	8	4	1048576	100	74.3
biology1	user3	24321	16	1	1048576	35.5	43.1
biology2	user3	26341	8	16	8388608	91.1	99.8
physics1	user2	29031	64	2	8388608	84.2	34.1
physics2	user4	30231	2	1	1048576	32.1	100

Figure 2: Overview of the creation of an integrated database from Darshan logs.

Table 1
Information extracted from Darshan logs.

Name	Description
ProgName	Name of the program
UserName	Name of the user
RunTime	Duration of the application
NumProcs	Number of processes
StripeCount	Number of OSTs used by the application
StripeSize	Amount of data written to an OSS per request
NumFile	Number of files used by the application
SeqIOPct	Percentage of sequential read/write requests
IOLess1K	Number of read/write requests less than 1K
IO1Kto100K	Number of read/write requests less than 100K
IOReadRequest	Number of read requests
IOWriteRequest	Number of write requests
IOBytesTotal	Total bytes read/written by the application
IOTimeTotal	Total read/write time used by the application
IOThroughputTotal	Total read/write throughput of the application

Darshan logs first must be transformed into a human-readable text format using the Darshan-parser utility [32]. After the transformation, the text file contains information, such as the program name, arguments, number of processes, and I/O activities for each I/O module (POSIX, MPIIO, and STDIO). Because this information is in a particular file format, it can be difficult to find the overall tendency of the applications.

To determine the overall I/O activities of the applications, we implemented a parser that extracts key information from the parsed Darshan text files and builds an integrated

Table 2
Result of analyzing stripe count.

StripeCount	Number of Executions	Percentage
1	1,275,869	99.317%
2	39	0.003%
3-4	62	0.005%
5-8	269	0.021%
9-16	6,850	0.533%
17-32	443	0.034%
33-64	374	0.029%
64-128	450	0.035%
129-256	287	0.022%
Total	**1284643**	**100%**

database. For the database engine, we selected SQLite [14] because it is lightweight, easy to use, and supports portability. By creating an integrated database, users can perform queries on various pieces of key information to determine the overall tendency of applications in the context of the entire HPC environment rather than for each application.

Table 1 lists the information extracted from Darshan log and inserted into the integrated database. While other information can be directly retrieved from the Darshan log, StripeCount (number of OSTs used by the application), IOTimeTotal (total I/O time), and IOThroughputTotal (aggregated throughput of I/O modules) must be computed. We used the following methods to compute that information.

- **StripeCount:** When Darshan collects I/O information, it checks whether the application uses Lustre file system and is compiled with the Lustre module enabled [33]. If the Lustre module is enabled, Darshan records LUSTRE_OST_ID, which is the OST_ID used in that specific I/O function call. While extracting the information, we track the number of OSTs involved in I/O during the application run and record the information in the integrated database.
- **IOTimeTotal and IOThroughputTotal:** Because Darshan collects the I/O duration based on the function call, many previous studies have used different approaches when calculating the total I/O time of an application. Wang et al. [37] calculated the I/O time by measuring the critical section because when an application uses the MPIIO module, many concurrent I/O processes can perform the I/O functions in parallel; thus, aggregating the duration of all I/O functions can be inaccurate. In this study, we use the approach used by Luu et al. [24], which measures the I/O time per process and uses the longest I/O time of all the processes. Using IOTimeTotal, we calculated I/OThroughputTotal, which is IOBytesTotal divided by IOTimeTotal.

3.2. Analysis of Darshan Logs

With the integrated database described in the previous section, we analyzed Lustre file system configuration used by users in the HPC environment. Tables 2 and 3 present the analysis results for stripe count and stripe size, respectively. As presented in both tables, we discovered that most users do not adjust Lustre file system configuration but instead use the default configuration. Table 2 indicates that 99.317% of executions use the default stripe count, which is 1 OST, while 256 OSTs are available in Cori. Similar to stripe count, Table 3 indicates that 99.948% of executions use the default stripe size, which is 1,048,576 (1 Megabyte). This analysis reveals that even in the HPC environment where users can exploit significantly more OSTs than in traditional computing environments, users do not adjust Lustre file system configuration. Thus, dynamic configuration control is necessary to fully exploit the I/O capabilities of the HPC environment.

Table 3
Result of analyzing stripe size.

StripeSize (Byte)	Number of Executions	Percentage
1,048,576	1,283,980	99.948%
4,194,304	1	0.000%
8,388,608	480	0.037%
16,777,216	162	0.013%
33,554,432	6	0.000%
50,331,648	4	0.000%
67,108,864	9	0.001%
100,663,296	1	0.000%
Total	**1,284,643**	**100%**

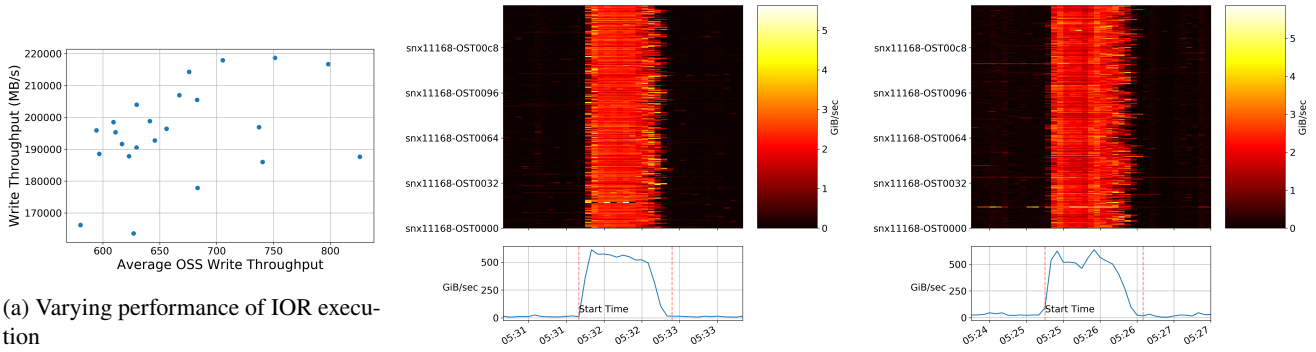
In addition, we also analyzed the number of unique applications among the 1,284,643 runs by querying the database for distinct application names. The results indicate that only 1,163 unique applications were executed during a two-month period. Because 1,284,643 executions occurred during that period, it can be inferred that this small number of applications were executed multiple times. Thus, using the information from previous executions, the performance of each additional execution can improve the performance because a high possibility that the application will run in the near future.

3.3. Analysis of I/O Interference on Multiple Applications

3.3.1. Cori

To analyze the I/O behaviors of the HPC environment when multiple applications are running, we analyzed the data from CORI. As a target application, we analyzed the IOR [6] benchmark, which is a widely used HPC I/O benchmark. In the system, the benchmark runs every day at the same time to monitor any problems and analyze the file system.

Figure 3a shows the performance of the IOR benchmark when multiple applications run simultaneously. The *y*-axis represents the write throughput, whereas the *x*-axis represents the average write throughput of OSSs when the application runs. To control the effect of different configurations, the figure only depicts the executions with an identical username, IOBytesTotal, SeqIOPct, and others. This suggests that the application remains identical and only the file system status changes. As shown in the figure, the write throughput is different because the status of the file system is different in each execution. The reason is that other applications also perform I/O operations on the file system.



(a) Varying performance of IOR execution

(b) Heatmap of a high performing execution

(c) Heatmap of a low performing execution

Figure 3: Performance of the IOR application in Cori.

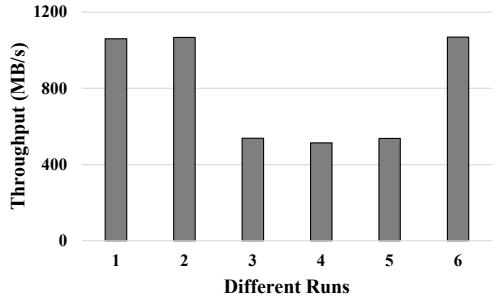


Figure 4: Performance of two FIO instances with a stripe count of 2.

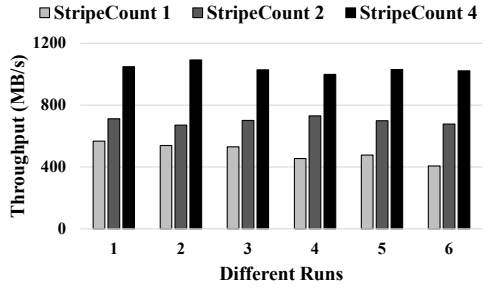


Figure 5: Performance of two FIO instances with varying stripe count.

To further analyze the effects of other applications, Figures 3b and 3c present the heatmap of OSSs when the I/O performance of IOR is high and low, respectively. Each horizontal line denotes an OSS. If the line is yellow, the OSS is heavily utilized. As indicated in the figures, the OSS activity is dominated by the I/O operations from the IOR. However, when the performance is relatively low (Figure 3c), some OSSs are used before the IOR is running. This is interference from a prior application, which results in the degradation of the overall application performance.

3.3.2. Local Environment

To analyze the effect of interference in a more controlled environment, we conducted another evaluation in a local environment. The environment had four OSSs, and each was equipped with two Samsung 850 pro solid-state drives (SSDs). For the application, we used the FIO [16] benchmark, which is a widely used I/O benchmark for a local environment.

Figure 4 shows the performance of the FIO benchmark when two instances of FIO run simultaneously. The figure presents the performance results of different executions to demonstrate performance variations. Both FIO instances are configured with stripe count of 2, utilizing two OSSs. As depicted in the figure, the performance of the application varies significantly in different runs. In an optimal situation, each instance takes a different pair of OSSs, and no overlapping occurs. For example, the first instance of FIO utilizes OSSs 1 and 2, whereas the second instance of FIO utilizes OSSs 3 and 4. However, when two instances share one or two OSSs, the performance is lower than 50% of that achieved in the optimal situation because interference occurs when multiple applications issue I/O operations to the same OSSs. Thus, the analysis results indicate that the performance degradation is severe when multiple applications are executed simultaneously without considering interference.

To identify the effects of interference when it is unavoidable, we conducted another analysis with the FIO benchmark, as presented in Figure 5. In this case, the FIO instance is configured with a stripe count of 4, and another FIO instance is configured with stripe count of 1, 2, and 4. Thus, all four OSSs are used continuously by the first FIO instance, and the second FIO instance uses one, two, and four OSSs. As shown in the figure, the performance is higher when interference occurs at all four OSSs, and the performance is lower when the number of overlapped OSSs is lower. Because FIO is an I/O intensive application, it is better to distribute the I/O requests across multiple OSSs rather than force a few OSSs to handle an overwhelming number of requests. However, the performance of the application is significantly lower than that of an isolated execution. This suggests that interference from multiple applications can affect the overall I/O performance in a shared parallel and distributed file system. Thus, it is crucial to dynamically control the configuration of the file system to reduce I/O interference.

4. Design and Implementation

In this section, we present the DCA-IO algorithm to control the Lustre file system configuration dynamically. DCA-

PROCEDURE 1 DCA-IO algorithm for initial execution.

```
1: New Application Request
2: /* check the number of processes */
3: currNumProcs = current number of processes
4: stripeCounts[]
5: stripeSizes[]
6: records[] = SELECT * FROM database WHERE numProcs == currNumProcs
7: for item in records
8:       if record.stripeCount is Unique
9:            stripeCounts.add(record.stripeCount)
10:      if record.stripeSize is Unique
11:           stripeSizes.add(record.stripeSize)
12:
13: stripeCount, stripesize = 0
14: for item in stripeCounts
15:      tempThroughput = aveThroughputOfItem
16:      if tempThroughput > stripeCount
17:           stripeCount = item
18: for item in stripeSizes
19:      tempThroughput = aveThroughputOfItem
20:      if tempThroughput > stripeSize
21:           stripeSize = item
22: lfs setstripe -c stripeCount -S stripeSize
```

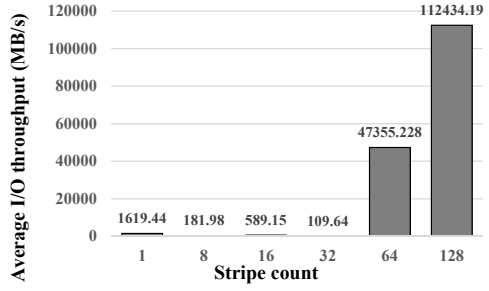


Figure 6: Average I/O throughput per stripe count when the number of processes is 1.

IO is divided into two parts: the application-specific configuration and interference optimization. As highlighted in the analysis section, it is important to optimize the configuration in terms of both application-specific behavior and interference to achieve optimal performance. The optimization of the application-specific behavior is achieved by the initial and recurring executions. In the initial execution, there is no prior knowledge of the incoming application, and the system must make a blind estimate without knowing the I/O behavior of the application. In the recurring execution, entries in the integrated database match the application name. Thus, we can employ the Darshan log from previous executions to optimize the configuration. While application name can be inconclusive to determine the I/O characteristics of the application, we use the I/O characteristics of previous execution with the identical name as inspecting input data or the code can induce a large overhead. [1] The optimization of the interference is comprised of single and multiple executions. The single execution deals with a situation in which an application is the only application that performs I/O operations in a system, whereas multiple executions indicate that more applications are already running and that interference may occur.

[1] Note that application executions with identical names can have different I/O behavior as the behavior can be impacted by input data, algorithms, and more. However, DCA-IO assumes that executions will have similar I/O behavior.

4.1. Application-specific Configuration Adjustment

4.1.1. Initial Execution

In the case of the initial execution, there is no information about the application because no Darshan log is available for the incoming application. Thus, it is impossible to make an adjustment based on application behavior. Instead, DCA-IO utilizes the number of processes because the user already specifies the number of processes by requesting resources. With the number of processes, DCA-IO uses existing Darshan logs of other applications in the same HPC environment. Although it is not guaranteed that the existing Darshan logs are related to the incoming application, DCA-IO makes a statistical guess based on the existing Darshan logs because they share the same hardware that is related to application performance [4].

Procedure 1 presents a simplified algorithm for handling a new application. When the application arrives, DCA-IO first records the number of processes provided by the user (line 3). Then, it uses the integrated database to select entries that have the same number of processes as the incoming application (line 6). Next, it extracts a unique stripe count from the entries and calculates the average I/O throughput per unique stripe count (lines 7-11). Finally, we set the stripe count of the application as the stripe count of the highest average I/O throughput(lines 14-17). DCA-IO repeats an identical algorithm to adjust the stripe size (lines 18-21).

For example, when a process is requested by the new incoming application, DCA-IO can refer to the entries that used one process from the integrated database. Figure 6 presents the average I/O throughput per unique stripe count from the integrated database. As illustrated in the figure, the unique stripe counts according to the integrated database are 1, 8, 16, 32, 64, and 128. Because the average I/O performance is highest for stripe count of 128, the stripe count will be set to 128. Thus, when no information on the incoming application exists, DCA-IO can make an educated adjustment based on Darshan logs from other applications that share identical hardware.

4.1.2. Recurring Execution

In the case of the recurring execution, Darshan logs with identical application names exist, and the I/O behaviors of the application can be utilized for the configuration adjustment. DCA-IO optimizes the configuration when the I/O behavior is available in two phases: the rule-based and heuristic phases.

Procedure 1 shows a simplified algorithm for handling recurring execution. In the rule-based phase, DCA-IO optimizes the configuration using the existing rules from many previous studies [27, 10]. If the I/O behavior of the application is *file-per-process*, where each file is used by a single process, the number of processes that can access a single file is one (lines 3-4). Thus, we first start with stripe count as 1 because using multiple stripe counts can increase the contention between multiple processes and the communication overhead. In the case of a *single shared file*, where multiple

PROCEDURE 2 DCA-IO algorithm for recurring execution.

```
1: Recurring Application Request
2: /* 2nd Execution - rule-based phase*/
3: if file-per-process
4:       stripeCount = 1
5: if shared file
6:       stripeCount = numIOProcs
7: stripeSize = 1M
8: lfs setstripe -c stripeCount -S stripeSize
9:
10: /* 3rd and more Executions - heuristic phase */
11: if IOthroughput > previousIOthroughput
12:       stripeCount = previousStripeCount * 2
13:       if stripeCount > maxAvailStripeCount
14:            stripeCount = maxAvailStripeCount
15: else
16:       stripeCount = previousStripeCount
17:
18: if stripeCount == previousStripeCount
19:       if IOthroughput > previousIOthroughput
20:            stripeSize = previousStripeSize * 2
21: else
22:       stripeSize = previousStripeSize
23: lfs setstripe -c stripeCount -S stripeSize
```

PROCEDURE 3 DCA-IO algorithm for interference adjustment.

```
1: stripeCount = Stripe Count from the previous procedure
2: stripeSize = Stripe Size from the previous procedure
3: startOffset, endOffset = 0
4: r = [] //Set of free OSSes
5: r.start = 0 //Start OSS
6: o //OSS with the lowest capacity
7:
8: /* 1st: Non-overlapping allocation */
9: iterate list of OSSs
10:       if set of free OSSs r exists
11:            startOffset = r.start
12:            endOffset = startOffset + stripeCount
13:            Update free flags for OSSs in r
14:            lfs setstripe -c stripeCount -S stripeSize -i startOffset
15:
16: /* 2nd: Partial overlapping allocation */
17: iterate list of OSSs
18:       if set of partially free OSSs r exists
19:            startOffset = r.start
20:            endOffset = startOffset + stripeCount
21:            Update free flags for OSSs in r
22:            lfs setstripe -c stripeCount -S stripeSize -i startOffset
23:
24: /* 3rd: Capacity based allocation */
25: iterate list of OSSs
26:       find the OSS with the lowest capacity o
27:       startOffset = index of o
28:       endOffset = startOffset + stripeCount
29:       lfs setstripe -c stripeCount -S stripeSize -i startOffset
```

processes can access shared files, we set stripe count as the number of processes participating in I/O because multiple processes can use a high stripe count (lines 5-6).

In both cases, DCA-IO sets the stripe size as 1M, which is the smallest possible size and the default configuration in Lustre file system for two reasons. First, Darshan does not record the sizes of the application requests but the size intervals to which the requests belong. Darshan classifies requests according to the range of the request size and records the number of requests that belong to each interval (e.g., 1K to 100K). Without knowing the specific request sizes of the application, using a large stripe size can create misaligned stripes in a file, which can degrade performance significantly [22, 18]. Second, Lustre suffers less from a small stripe size than a large stripe size. According to previous studies [22, 26], Lustre already aggregates small striped requests until they match the stripe alignment that decreases the overhead of using a small stripe size. Thus, rather than starting from a large stripe size, DCA-IO sets the stripe size as the minimum and gradually increases the size during the heuristic phase.

In the heuristic phase, DCA-IO increases the stripe count linearly until the performance decreases or the stripe count reaches the maximum available number of OSTs in the system (lines 11-16). The reasoning for this algorithm differs for different access patterns. In the case of *file-per-process*, the I/O performance of each file is bound to the maximum performance of an OST because the stripe count is set to 1 during the rule-base phase. However, if the application generates a large amount of I/O rapidly, the maximum performance of a single OST can be insufficient to handle the I/O requests for a file. Thus, DCA-IO tests a larger stripe count to determine whether a limited stripe count bounds the application performance. In the case of *shared-file*, multiple processes can access the same file concurrently. Thus, increasing the stripe count beyond the number of processes can mitigate the bottleneck.

In the case of stripe size, our proposed algorithm increases the stripe size only if the stripe count is the same as the previous run (lines 18-22). This isolates the effect of the stripe size from the varying stripe counts. DCA-IO then increases the stripe size until the performance decreases because the large stripe size can be beneficial to applications that issue large-sized requests. Thus, by increasing both the stripe count and stripe size, DCA-IO covers most of the configuration spaces and dynamically improves the application performance.

4.2. Interference Adjustment

When the existing Lustre file system allocates an OSS to an application, the Lustre file system randomly chooses the OSS as a starting OSS to handle the application requests. If the application needs more than one OSS, Lustre file system randomly chooses the starting OSS and then chooses the following OSSs. For example, if there are four OSSs in the system and the application needs three OSSes, Lustre first randomly selects an OSS out of four OSSes as a starting OSS. If the randomly chosen starting OSS is the second OSS, the following OSSs are the third and fourth servers.

If a certain OSS is used intensively compared with other OSSs, it can create both temporal and spatial imbalance among OSSs [29]. When I/O requests from multiple applications can be converged to a few OSSs, the performance of applications decreases due to the increased I/O time, resulting in a temporal imbalance. When a single OSS can run out of capacity and cannot to handle any new write requests, the overall I/O parallelism of the system decreases, resulting in spatial imbalance.

To solve the temporal and spatial imbalance, our proposed scheme first searches for the OSSs that are free (temporal imbalance). If all OSSs are being utilized by applications, our proposed scheme searches the OSSs with the low-

est capacity (spatial imbalance). Procedure 3 shows a simplified algorithm for our proposed scheme. When an application arrives, DCA-IO first determines the stripe count and size using algorithms from Procedure 1 or Procedure 2 (lines 1-2). Then, with the optimal stripe count and size, DCA-IO considers the interference and chooses one of three allocation strategies: non-overlapping, partially overlapping, or capacity-based allocations.

In the non-overlapping allocation, DCA-IO first attempts to examine whether a set of free OSSs (r) can be assigned to the application (line 7). DCA-IO iterates the list of OSSs to obtain a consecutive set of OSSs large enough to accommodate the requested number of OSSs. If a set of free OSSs meets the conditions, we retrieve the OSS range and calculate the end offset by adding the stripe count of the application and the starting offset of the free OSSs (lines 8-9). Then, DCA-IO marks free flags of selected OSSs as not free (line 10). Finally, the proposed scheme sets the stripe count, size, and starting offset for the application (line 11). Thus, the proposed scheme can choose a set of OSSs that are not used by any application, and the I/O request from the application is not interfered by other applications. If no such range is found, DCA-IO proceeds to the second phase, which is partial overlapping.

If DCA-IO cannot find a set of OSSs that are free, it attempts to find a set of partially free OSSs and distributes the I/O requests to OSSs. To this end, DCA-IO iterates the list of OSSs and identifies a set of OSSs that has the largest number of free OSSs (lines 14-15). This strategy minimizes interference from other applications by reducing the number of shared OSSs. With the found set of partially free OSSs, similar to the case of non-overlapping allocation, DCA-IO updates the free flag of the selected OSSs (line 18) and sets the stripe count, size, and starting offset (line 19).

If no free OSSs are available because all OSSs in the system is used by other applications, DCA-IO performs capacity-based allocation which is an allocation policy to balance the capacity of the OSSs. To balance out the capacities among the OSSs, it checks the remaining capacities of the OSSs and chooses the OSS with the lowest capacity (lines 22-23). Similar to other phases, DCA-IO sets the stripe count, size, and starting offset (line 26). In summary, DCA-IO first attempts to identify a set of completely isolated free OSSs (non-overlapping allocation). If the set does not exist, it attempts to identify a set of OSSs with the highest number of free OSSs (partial overlapping allocation). Finally, if all the OSSes are used by other applications, it attempts to use the OSSs with the lowest capacity, balancing out the capacity among OSSs for future applications (capacity-based allocation). Thus, DCA-IO can dynamically adjust the configuration based on the isolated application performance and reduce the performance interference caused by other applications.

Table 4
Comparison between TAPP-IO and DCA-IO.

	TAPP-IO [27]	DCA-IO
File-based	✓	✓
Rule-based	✓	✓
Dynamic		✓
Interference		✓
Capacity-aware		✓

5. Evaluation

5.1. Single Application

5.1.1. Local Environment

For the evaluation, we used a six node Lustre setup having Intel i7-4790 (3.6 GHz) having four physical cores, eight cores with hyper-threading, and 8 GB of memory. We used one node for the client server, one node for the MDS, and four nodes for the OSSs. We used a Samsung 850 pro SSD for storage. Each node had two SSDs, one for the operating system and another for Lustre file system. We used two Intel 2P X520 10G network adapters per node. Because two 10 G network adapters can support a bandwidth of up to 2.5 GB, the I/O performance from the client was bottlenecked by the storage devices and not the network. We compared the performance of the Lustre default parameters, TAPP-IO (state of art storage allocation scheme) [27]), and DCA-IO. Table 4 shows the comparison between TAPP-IO and DCA-IO. As shown in the table, similar to DCA-IO, TAPP-IO is a rule-based scheme that determines stripe count and stripe size based on file access type (file-per-process and shared file). In contrast, DCA-IO supports dynamic adjustment that finds stripe count and stipe size through a heuristic phase. In addition, it also considers interference between multiple applications and capacity-aware data placement. We report the experimental results were averaged over five runs for default, TAPP-IO, and DCA-IO.

For the microbenchmark, we ran the FIO benchmark [16] performing sequential and random writes. The FIO benchmark creates a separate file for each process and uses the POSIX I/O module. We configured the FIO benchmark to issue 8 GB write operations using one to eight threads with a 1 MB request size and buffered I/O.

In the case of sequential writes, as indicated in Figure 7, DCA-IO improves the performance by up to 75% compared with both the default Lustre configuration and TAPP-IO. The performances achieved by the default Lustre configuration and TAPP-IO is comparable because TAPP-IO uses the default configuration in the case of the *file-per-process*. The performance improvement is greater when the number of processes is smaller because both the default Lustre configuration and TAPP-IO set the stripe count to 1 (i.e., 1 OSS). Because FIO issues a large number of I/O requests, the I/O performance is bottlenecked by the maximum I/O performance of the single OSS. Thus, by increasing the stripe count, DCA-IO can improve the performance beyond the maximum performance of a single OSS. In the case of a large number of

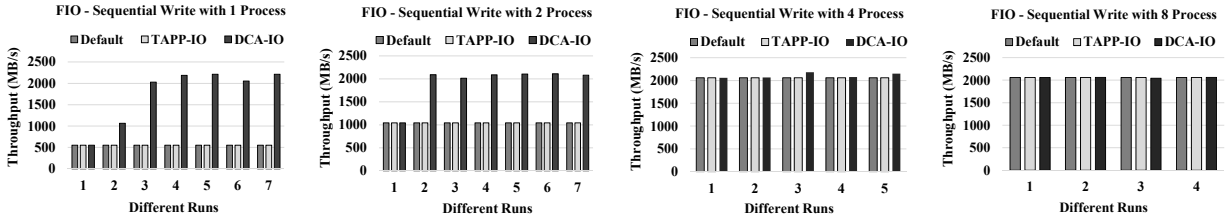


Figure 7: FIO sequential write performance.

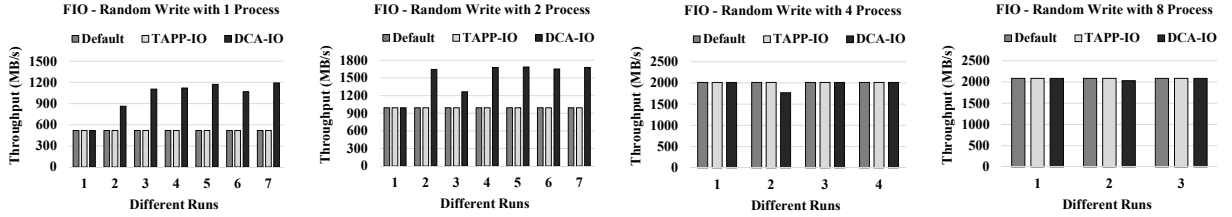


Figure 8: FIO random write performance.

processes, all four OSSs were used even with a stripe count of 1 because each process creates a file allocated to different OSSs. Thus, the performance of FIO reaches the maximum performance of all OSSs using both the default Lustre configuration and TAPP-IO.

In the case of random writes, as indicated in Figure 8, DCA-IO improves the performance by 56% compared with both the default Lustre configuration and TAPP-IO. Similar to sequential writes, the performance improvement is more evident at a lower number of processes and the reason is the same. However, because the performance for random writes is inherently inferior to that for sequential writes, the performance improvement is smaller in the former than in the latter case.

For the macrobenchmark, we used Parallel I/O Kernel (PIOK) [7] developed by NERSC. PIOK is a collection of I/O kernels from three HPC applications: VPIC, GCRM, and VORPAL. Thus, VPCI-IO, GCRM-IO, and VORPAL-IO do not perform computation tasks but only issue I/O operations for synthetic data structures. PIOK is implemented to utilize both the HDF5 file format [12] and the H5Part data interface [2]. We configured each benchmark to use collective I/Os where each process calls collective I/O functions to aggregate multiple I/O requests into collective I/O requests.

In the case of VPIC-IO, as shown in Figure 9, DCA-IO improves the performance by up to 70% and 45%, respectively, compared with the default Lustre configuration and TAPP-IO. Similar to the result from the FIO benchmark, the performance gain from a smaller number of processes results from the increased number of stripe counts. Because TAPP-IO uses the number of processes as the stripe count, the performance at a low number of processes is related to the limited number of OSSs. For a high number of processes, the performance of DCA-IO is higher than that of TAPP-IO owing to the stripe alignment. Because DCA-IO gradually increases the stripe size from 1M, it can find the optimal stripe size without causing stripe misalignment. Note that in the case of the second run in the one, two, and four processes, the performance of DCA-IO decreases from the first run. Because DCA-IO uses rule-based configuration adjustment, the adjusted configurations cannot be optimal, compared to the adjusted configurations from the initial execution. However, the performance becomes comparable to or exceeds that of the first run due to the second heuristic phase of DCA-IO.

In the cases of GCRM-IO and VORPAL-IO, as shown in Figures 10 and 11, DCA-IO improves performance by up to 58% and 52% compared with the default Lustre configuration, and by up to 48% and 51% compared with TAPP-IO, respectively. Similar to VPIC-IO, the performance of DCA-IO is significantly better than that of TAPP-IO due to the effect of the stripe count. In addition, owing to the stripe misalignment, the performance of TAPP-IO is lower than that of DCA-IO.

5.1.2. Cori

To verify the effectiveness of DCA-IO in a complex and large environment, we conducted the experiment in Cori. For the evaluation, we used 1, 4, 16, and 64 computation nodes from Cori. Each compute node was equipped with two 16-core Intel Haswell CPUs (2.3 GHz) and 128 GB of memory. For storage, the Lustre file system of Cori had 6 MDSs and 256 OSTs. Both the compute and Lustre nodes were connected with Infiniband. For a benchmark, we only used VPIC-IO from PIOK [7] because the other two workloads exhibited similar I/O behaviors. To evaluate our scheme with diverse application behavior, we used both independent and collective I/O modes. The experimental results are the average values of five runs. In addition, the average I/O traffic at the time of the experiment was less than 5 percent of the maximum bandwidth of Cori system.

In the case of VPIC-IO in Cori, as shown in Figures 12 and 13, DCA-IO improves performance by up to 37% and 50% in independent I/O and collective I/O, respectively, compared with TAPP-IO. Compared to the results from a small Lustre setup, the performance improvement on Cori is less for two main reasons. First, because the experiments in Cori

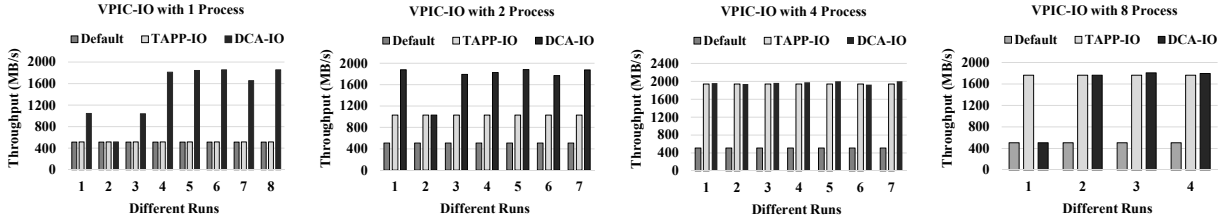


Figure 9: VPIC-IO performance.

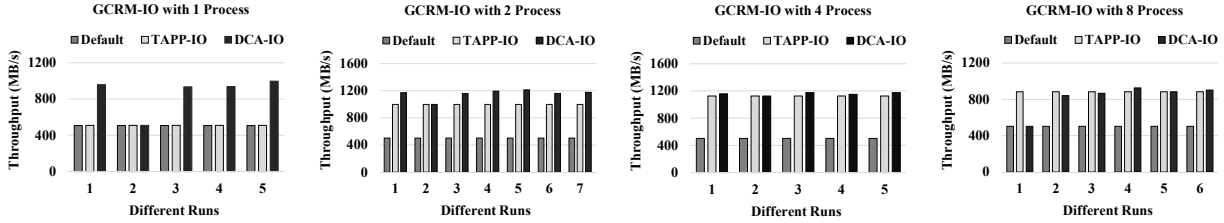


Figure 10: GCRM-IO performance.

already have a high number of processes, TAPP-IO, which sets the stripe count to the number of processes, already utilizes a sufficient number of OSSs. Second, because many users share the same HPC environment, there can be many interferences from the I/O activities of other users. Owing to the other users, our application cannot utilize the full bandwidth of the network. Because the benefits from DCA-IO are more evident when the I/O performance of applications is better than the maximum performance of the used OSSs, the potential performance improvement can be overshadowed by various resource contentions in a complex HPC environment. However, DCA-IO can improve the performance of different I/O behaviors such as independent and collective I/O modes. Thus, we have verified that DCA-IO could be beneficial in small isolated environments as well as in large production scale environments.

5.2. Multiple Applications

To evaluate DCA-IO when multiple applications are running simultaneously, we performed an evaluation using the aforementioned FIO [16], VPIC-IO, GCRM-IO, and VORPAL-IO benchmarks from PIOK [7]. We configured each application to run with four processes because four processes exhibit the best performance. To evaluate our scheme in various performance interference scenarios, we disabled DCA-IO for FIO and manually set the stripe count of FIO to 1, 2, and 4 and we enabled DCA-IO for VPIC-IO, GCRM-IO, and VORPAL-IO. Thus, we created scenarios where an I/O-intensive application (FIO) with various stripe counts runs in the system, and another application (VPIC-IO, GCRM-IO, or VORPAL-IO) is submitted to the system. We compared the performance of the PIOK benchmark using the default Lustre configuration, DCA-IO without interference optimization, and DCA-IO with interference optimization (i.e., fully optimized DCA-IO). All experimental results are the average values of five runs.

Figure 14a presents the performance of the VPIC-IO benchmark from the PIOK when the VPIC-IO and FIO benchmarks run simultaneously. As shown in the figure, the DCA-IO with interference optimization performs similarly to DCA-IO without interference optimization and improves performance by up to 263% compared with the default configuration. The performance is similar in both versions of DCA-IO because the benefit of using all four OSSs is greater than using a small number of OSSs and avoiding interference. Thus, both versions of DCA-IO set the stripe count of VPIC-IO as 4, utilizing all OSSs in the system. In both versions of DCA-IO, the performance of VPIC-IO is best when the stripe count of FIO is 1. This suggests that the performance is best when the fewest OSSs are shared between two applications. In contrast, when a default configuration is used, both FIO and VPIC-IO use a single OSS, and the application performance is bounded by the maximum performance of a single OSS.

Figures 14b and 14c present the performance of GCRM-IO and VORPAL-IO when FIO is running. In the case of

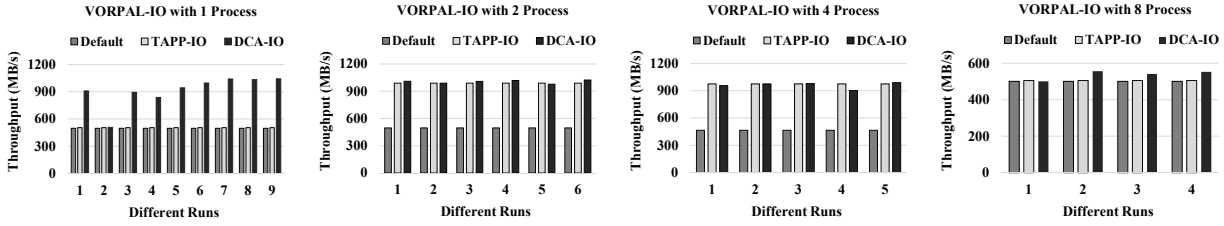


Figure 11: VORPAL-IO performance.

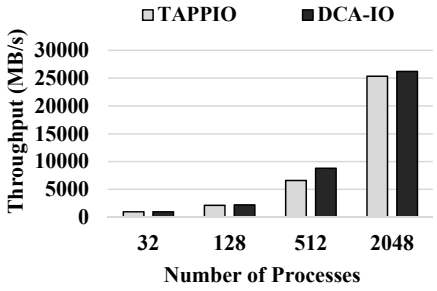


Figure 12: VPIC-IO performance in Cori using independent-I/O.

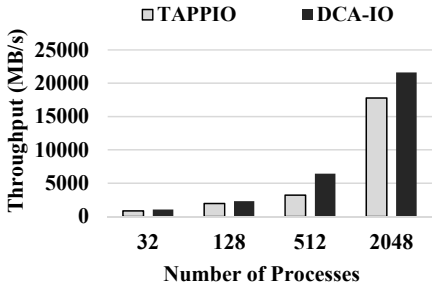


Figure 13: VPIC-IO performance in Cori using collective-I/O.

GCRM-IO, DCA-IO with interference optimization improves performance by up to 241% and 31%, compared with the default Lustre configuration and DCA-IO without interference optimization, respectively. For VORPAL-IO, DCA-IO with interference optimization improves performance by up to 252% and 53% compared with the default Lustre configuration and DCA-IO without interference optimization, respectively. In contrast to VPIC-IO, for both GCRM-IO and VORPAL-IO, DCA-IO with interference optimization performs better than DCA-IO without interference optimization. This is because DCA-IO without interference optimization uses all four OSSs for both GCRM-IO and VORPAL-IO, while DCA-IO with interference optimization uses stripe count of 2. Because the performance gain from increasing the stripe count in both applications is smaller than that of VPIC-IO, it is more beneficial to reduce the number of stripe count and utilize the OSSes that are not used by another application (FIO). However, when the stripe count of FIO is 4 and interference is unavoidable, the performance of both versions of DCA-IO remains identical. Thus, the evaluation results indicate that when multiple applications run simultaneously in the system, DCA-IO with interference optimization can improve their performance by reducing the interference from other applications by choosing free OSSs.

6. Related Work

In this section, we present previous studies and compare them to DCA-IO to show how DCA-IO differs from the previous studies. There have been many approaches to improving the I/O performance of applications in distributed computing systems with complex storage architecture. Previous works [40, 39, 5] proposed testing-based adjustment schemes that attempt to find the optimal configuration through performance modeling. In addition, other works [13, 3] proposed history-based adjustment schemes that find the optimal configuration based on the history of applications. Another work [27] proposed a rule-based adjustment scheme that adjusts the configuration based on the rules such as the number of files. Finally, several works [20, 11] alleviated performance interference between applications by improving communication.

6.1. Testing-based Adjustment

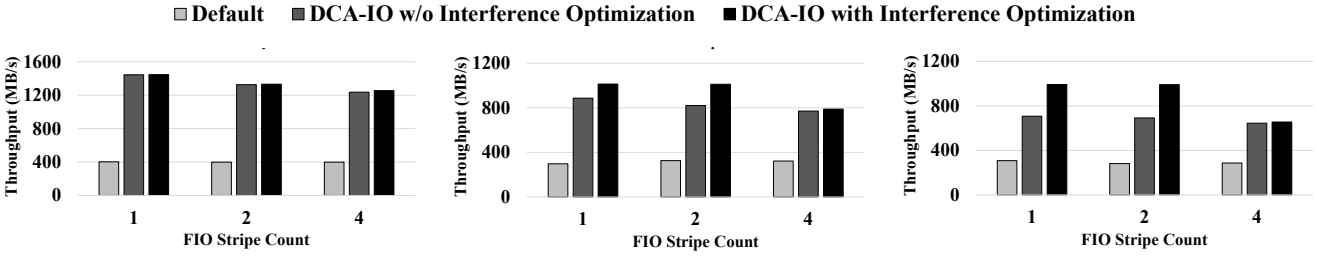
Several studies have been conducted on the improvement of application performance by modeling the I/O behaviors of applications. Yu et al. [40] classified applications into a few categories based on their I/O behaviors. Then, they found the optimal configuration setting for each distinct I/O behavior by performing extensive testing. Finally, they used the optimal configuration from the testing for each I/O behavior category. You et al. [39] proposed a mathematical model based on the queuing theory. Then, they performed experiments for each model in a separate environment to find the optimal configuration. H5Evolve [5] utilized a genetic algorithm to search for the best configuration. It simplifies multiple configurations into a simplified and computable space. Then, it uses the genetic algorithm to find the best configuration from the configuration space. Our study is similar to these studies in terms of investigating the I/O behaviors of applications and optimizing the performance based on I/O behaviors. However, DCA-IO does not require testing on various configurations prior to the application run.

6.2. History-based Adjustment

Several studies have been conducted on improving the I/O performance of applications by utilizing the previous history of applications. Gainaru et al. [13] stored the I/O behaviors and used the history of each application during scheduling to minimize the interference between applications. Behzad et al. [3] extracted I/O patterns from applications and found optimal configurations for each pattern. Then, they stored the optimal configuration for each pattern in a database. Our study is similar to these investigations in terms of optimizing the configurations based on the previous executions. However, DCA-IO can improve performance when no information on the I/O behavior is available, by using the existing system logs in the same HPC environment.

6.3. Rule-based Adjustment

Studies have been conducted on the selection of configurations based on a set of rules. TAPP-IO [27] is the most closely related to our algorithm in that the Lustre file system settings are optimized. Researchers [27] proposed a set of rules based on the number of files and processes. They evaluated their rule-based algorithm in a large HPC environment and verified that a rule-based configuration adjust-



(a) VPIC-IO (b) GCRM-IO (c) VORPAL-IO

Figure 14: VPIC-IO, GCRM-IO, and VORPAL-IO performance when running simultaneously with FIO.

ment scheme could improve performance in many complex HPC environments. Our study is similar to this research in terms of optimizing the configurations based on the set of rules. However, DCA-IO dynamically improves performance based on the previous runs because the optimal rules may vary according to the I/O behavior of the application.

6.4. Alleviating Performance Interference

Several studies have been conducted on the mitigation of I/O performance interference in the HPC environment. Lofstead et al. [20] analyzed an HPC system and presented the negative effects of simultaneous application execution on shared resources in the system. Then, to alleviate the I/O contention on a shared file system, they proposed an algorithm that enables applications to implement the I/O request with minimum interference through communication between applications. Dorier et al. [11] analyzed the performance interference in the I/O stack layer and designed a framework consisting of strategies for alleviating the influence of I/O interference. The framework uses MPI routines for communication between applications running in parallel. Our study is similar to these investigations in terms of the characterization of multiple cases of I/O performance interference and the alleviation of interference. However, DCA-IO minimizes the performance degradation by autonomously and dynamically adjusting the file system configurations by using various system logs in the HPC system.

7. Conclusion

In this paper, we proposed a dynamic distributed file system configuration adjustment scheme called DCA-IO to improve the I/O performance of applications and mitigate I/O performance interference in the HPC environment. To this end, we first analyzed the I/O behaviors of applications executed in Cori. The analysis results indicate that only a limited number of programs were executed extensively and that most of the executions used the default Lustre file system configuration. To improve the I/O performance of the applications by adjusting the Lustre file system configuration, DCA-IO uses existing system logs from the HPC environment and gradually improves performance using the rules and history of the program executions. Furthermore, DCA-IO reduces the contention for shared storage resources by dynamically adjusting the allocation policy. Finally, we evaluated DCA-IO using the FIO and PIOK benchmarks on small- and large-scale HPC environments using the Lustre file system. Our evaluation indicates that the use of DCA-IO for an independent application can improve the performance by up to 75% and 50% in small- and large-scale HPC environments, respectively. For the execution of simultaneous applications, the evaluation indicates that DCA-IO can improve performance by up to 263% and 53% compared with the default Lustre configuration and DCA-IO without interference optimization, respectively. For the future work, we plan to improve the performance heuristic phase of DCA-IO by introducing well-known optimization algorithms [1].

8. Declaration

Data Availability Raw data were generated at NERSC. Derived data supporting the findings of this study are available from the corresponding author on request.

Funding This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2021R1C1C1010861). This work was supported in part by the Korea Institute for Advancement of Technology (KIAT) grant funded by Korea government (MOTIE) (P0012724, The Competency Development Program for Industry Specialist). This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center. This study was financially supported by Seoul National University of Science and Technology.

Ethical approval Ethical approval was not required for this research.

Informed consent All the authors listed have approved the manuscript for publication.

Author contributions Sunggon Kim contributed the paper through conceptualization, methodology, software, and writing. Alex Sim, Kesheng Wu, and Suren Byna contributed the paper through conceptualization, discussion, and supervision. Yongseok Son contributed the paper through conceptualization, discussion, writing, and supervision (Corresponding Author: Yongseok Son).

References

[1] Abualigah, L., Yousri, D., Abd Elaziz, M., Ewees, A.A., Al-Qaness, M.A., Gandomi, A.H., 2021. Aquila optimizer: a novel meta-heuristic optimization algorithm. Computers & Industrial Engineering 157, 107250.

[2] Adelmann, A., Gsell, A., Oswald, B., Schietinger, T., Bethel, W., Shalf, J., Siegerist, C., Stockinger, K., 2007. Progress on h5part: a portable high performance parallel data interface for electromagnetics simulations, in: 2007 IEEE Particle Accelerator Conference (PAC), IEEE. pp. 3396–3398.

[3] Behzad, B., Byna, S., Snir, M., et al., 2015a. Pattern-driven parallel i/o tuning, in: Proceedings of the 10th Parallel Data Storage Workshop, ACM. pp. 43–48.

[4] Behzad, B., Byna, S., Wild, S.M., Snir, M., et al., 2015b. Dynamic model-driven parallel i/o performance tuning, in: Cluster Computing (CLUSTER), 2015 IEEE International Conference on, IEEE. pp. 184–193.

[5] Behzad, B., Luu, H.V.T., Huchette, J., Byna, S., Aydt, R., Koziol, Q., Snir, M., et al., 2013. Taming parallel i/o complexity with auto-tuning, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM. p. 68.

[6] Benchmark, I., 2020. https://asc. llnl. gov/sequoia/benchmarks/ior summary v1. 0. pdf. Accessed January 5.

[7] Byna, S., Howison, M., 2015. Parallel i/o kernel (piok) suite.

[8] Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R., 2011. Understanding and improving computational science storage access through continuous characterization. ACM Transactions on Storage (TOS) 7, 8.

[9] Chan, A., Gropp, W., Lusk, E., 2008. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. Scientific Programming 16, 155–165.

[10] for Computational Sciences, T.N.I., . I/o and lustre usage. URL: https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips.

[11] Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S., 2014. Calciom: Mitigating i/o interference in hpc systems through cross-application coordination, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE. pp. 155–164.

[12] Folk, M., Cheng, A., Yates, K., 1999. Hdf5: A file format and i/o library for high performance computing applications, in: Proceedings of supercomputing, pp. 5–33.

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

[14] Hipp, D.R., 2000. Sqlite. URL: https://www.sqlite.org/index.html.

[15] Howison, M., 2010. Tuning hdf5 for lustre file systems .

[16] J.Axboe, 1998. Fiobenchmark. http://freecode.com/projects/fio.

[17] Kim, S., Sim, A., Wu, K., Byna, S., Wang, T., Son, Y., Eom, H., 2019. Dca-io: A dynamic i/o control scheme for parallel and distributed file systems., in: CCGRID, pp. 351–360.

[18] Liao, W.k., Ching, A., Coloma, K., Choudhary, A., Ward, L., 2007. An implementation and evaluation of client-side file caching for mpi-io, in: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, IEEE. pp. 1–10.

[19] Lockwood, G.K., Snyder, S., Wang, T., Byna, S., Carns, P., Wright, N.J., 2018. A year in the life of a parallel file system, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE. pp. 931–943.

[20] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., Wolf, M., 2010. Managing variability in the io performance of petascale storage systems, in: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE. pp. 1–12.

[21] Lustre, a. Lustre* software release 2.x - operations manual .

[22] Lustre, A., b. Lustre technical white paper .

[23] Luu, H., Behzad, B., Aydt, R., Winslett, M., 2013. A multi-level approach for understanding i/o activity in hpc applications, in: 2013 IEEE International Conference on Cluster Computing (CLUSTER), IEEE. pp. 1–5.

[24] Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., Yao, Y., 2015. A multiplatform study of i/o behavior on petascale supercomputers, in: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, ACM. pp. 33–44.

[25] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L., 2007. The new ext4 filesystem: current status and future plans, in: Proceedings of the Linux symposium, pp. 21–33.

[26] Minich, M., Di, W., Shipman, G.M., Canon, S.O.S., 2008. The lustre center of excellence at ornl .

[27] Neuwirth, S., Wang, F., Oral, S., Bruening, U., 2017. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance, in: Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on, IEEE. pp. 604–613.

[28] Odajima, T., Kodama, Y., Tsuji, M., Matsuda, M., Maruyama, Y., Sato, M., 2020. Preliminary performance evaluation of the fujitsu a64fx using hpc applications, in: 2020 IEEE International Conference on Cluster Computing (CLUSTER), IEEE. pp. 523–530.

[29] Patel, T., Byna, S., Lockwood, G.K., Wright, N.J., Carns, P., Ross, R., Tiwari, D., 2020. Uncovering access, reuse, and sharing characteristics of {I/O-Intensive} files on {Large-Scale} production {HPC} systems, in: 18th USENIX Conference on File and Storage Technologies (FAST 20), pp. 91–101.

[30] Schwan, P., et al., 2003. Lustre: Building a file system for 1000-node clusters, in: Proceedings of the 2003 Linux symposium, pp. 380–386.

[31] Shende, S.S., Malony, A.D., 2006. The tau parallel performance system. The International Journal of High Performance Computing Applications 20, 287–311.

[32] Snyder, S., Carns, P., Harms, K., Latham, R., Ross, R., 2016a. Performance evaluation of Darshan 3.0. 0 on the Cray XC30. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[33] Snyder, S., Carns, P., Harms, K., Ross, R., Lockwood, G.K., Wright, N.J., 2016b. Modular hpc i/o characterization with darshan, in: Extreme-Scale Programming Tools (ESPT), Workshop on, IEEE. pp. 9–17.

[34] Snyder, S., Carns, P., Latham, R., Mubarak, M., Ross, R., Carothers, C., Behzad, B., Luu, H.V.T., Byna, S., et al., 2015. Techniques for modeling large-scale hpc i/o workloads, in: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, ACM. p. 5.

[35] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., Peck, G., 1996. Scalability in the xfs file system., in: USENIX Annual Technical Conference.

[36] Tang, H., Byna, S., Tessier, F., Wang, T., Dong, B., Mu, J., Koziol, Q., Soumagne, J., Vishwanath, V., Liu, J., et al., 2018. Toward scalable and asynchronous object-centric data management for hpc, in: 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE. pp. 113–122.

[37] Wang, T., Snyder, S., Lockwood, G., Carns, P., Wright, N., Byna, S., 2018. Iominer: Large-scale analytics framework for gaining knowledge from i/o logs, in: 2018 IEEE International Conference on Cluster Computing (CLUSTER), IEEE. pp. 466–476.

[38] Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D., Maltzahn, C., 2006. Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association. pp. 307–320.

[39] You, H., Liu, Q., Li, Z., Moore, S., 2011. The design of an auto-tuning i/o framework on cray xt5 system, in: Cray User Group meeting (CUG 2011).

[40] Yu, W., Vetter, J.S., Oral, H.S., 2008. Performance characterization and optimization of parallel i/o on the cray xt, in: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, IEEE. pp. 1–11.

Sunggon Kim is an assistant professor in the department of Computer Science at Seoul National University of Science and Technology (SeoulTech) since 2022. He received his B.S. degree in Computer Science from University of Wisconsin-Madison, Madison, USA, and Ph.D. degree from Seoul National University in 2015 and 2021, respectively. He was an intern at Lawrence Berkeley National Laboratory, California, USA, in 2018, 2019 and 2020. His research interests are file systems, cloud computing, distributed systems, and operating systems.

Alex Sim is currently a Senior Computing Engineer at Lawrence Berkeley National Laboratory. He authored and co-authored over 300 technical publications, and released a few software packages under open source license. His current research and development activities include data modeling, data analysis methods, learning models, distributed resource management, and high performance data systems. He is a senior member of IEEE.

Kesheng Wu is a Senior Scientist at Lawrence Berkeley National Laboratory. He works extensively on data management, data analysis, and scientific computing topics. He is the developer of a number of widely used algorithms including FastBit bitmap indexes for querying large scientific datasets, Thick-Restart Lanczos (TRLan) algorithm for solving eigenvalue problems, and IDEALEM for statistical data reduction and feature extraction.

Suren Byna received his Ph.D. degree in 2006 in Computer Science from Illinois Institute of Technology, Chicago. He is a Staff Scientist in the Scientific Data Management (SDM) Group in CRD at Lawrence Berkeley National Laboratory (LBNL). He works on optimizing parallel I/O and on developing systems for managing scientific data. He is the PI of the ECP funded ExaIO and ExaHDF5 projects, and various projects on managing scientific data.

Yongseok Son received his B.S. degree from Ajou University in 2010, and his M.S. and Ph.D. degrees from Seoul National University in 2012 and 2018, respectively. He was a postdoctoral research associate at University of Illinois at Urbana-Champaign. Currently, he is an assistant professor in Department of Computer Science and Engineering, Chung-Ang University. His research interests are operating, distributed, and database systems.