This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

Optimizing Parallel I/O Accesses through Pattern-Directed and Layout-Aware Replication

Shuibing He, Yanlong Yin, Xian-He Sun *IEEE Fellow*, Xuechen Zhang, Zongpeng Li

Abstract—As the performance gap between processors and storage devices keeps increasing, I/O performance becomes a critical bottleneck of modern high-performance computing systems. In this paper, we propose a pattern-directed and layout-aware data replication design, named PDLA, to improve the performance of parallel I/O systems. PDLA includes an HDD-based scheme *H-PDLA* and an SSD-based scheme *S-PDLA*. For applications with relatively low I/O concurrency, H-PDLA identifies access patterns of applications and makes a reorganized data replica for each access pattern on HDD-based servers with an optimized data layout. Moreover, to accommodate applications with high I/O concurrency, S-PDLA replicates critical access patterns that can bring performance benefits on SSD-based servers or on HDD-based and SSD-based servers. We have implemented the proposed replication scheme under MPICH2 library on top of OrangeFS file system. Experimental results show that H-PDLA can significantly improve the original parallel I/O system performance and demonstrate the advantages of S-PDLA over H-PDLA.

Index Terms—Parallel I/O; I/O optimization; data replication; data reorganization; data access pattern

1 Introduction

Supercomputers are moving from petascale towards exascale in the coming decade. Although the rapid development of semiconductor technology made the processor speed to increase exponentially in the last decade, the advancements of data input/output (I/O) systems and storage devices did not keep pace with that of the computing power. This unbalanced technology advance between processors and storage devices leads to the so-called I/O-wall problem [1].

At the same time, applications on high-performance computing (HPC) domains have increasingly massive data requirements [2], putting even more pressure on already saturated I/O systems. For instance, in astronomy, giant radio telescopes will store the observation images continuously into storage systems for further analysis. The telescopes may generate data at a rate of several gigabytes to even petabytes per second [3].

To match the massive data requirements of scientific applications, HPC centers have widely deployed parallel I/O systems to provide data services. By scaling out the storage system with more file servers, supercomputers can achieve scalable I/O bandwidth and storage capacity. Unfortunately, while parallel I/O systems deliver decent peak performance for many data access patterns, such as large I/O requests, they fail to perform well for specific access patterns, such as non-contiguous small I/O requests [1], [4].

There are many I/O optimization techniques developed to improve I/O system performance, such as data sieving [5], List I/O [6], DataType I/O [7], and Collective I/O [5]. However, optimizing I/O performance is an error-prone and time-consuming task, especially for applications with complex I/O behaviors. I/O performance is application dependent, and a general I/O system needs to be adjustable to different applications [8], [9]. This raises the desired property of our proposed solution: I/O optimization should consider both application and system characteristics and be adaptive for different applications.

In this paper, we propose a pattern-directed and layout-aware data replication approach, named PDLA, to boost the performance of HDD-based parallel I/O systems, which nowadays are still the dominant storage platform for HPC applications. The "pattern" means how the file is accessed by an application and the "layout" means how the file data is distributed across multiple file servers. PDLA is motivated by three observations. First, scientific applications usually exhibit data access patterns. It is valuable and feasible to make use of application's pattern information for I/O optimizations. Second, contiguous data access is usually preferable to obtain higher I/O performance for both hard disk drives (HDDs) and solid state disks (SSDs). Third, data layout in parallel file systems can largely influence I/O system performance.

Based on these three facts, we first propose an HDD-based pattern-directed and layout-aware replication scheme, H-PDLA, to improve the parallel I/O system performance on HDD-based servers (HServers). H-PDLA includes two major optimizations. In *the "pattern-directed" (PD) replication policy*, the system makes a reorganized data

- *S. He is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310058, China. E-mail: heshuibing@zju.edu.cn*
- *Y. Yin is with the Intelligent Computing System Research Center, Institute of Artificial Intelligence, Zhejiang Lab, Hangzhou, Zhejiang 311100, China. E-mail: yyin@zhejianglab.com*
- *X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois 60616, USA. E-mail: sun@iit.edu*
- *X. Zhang is with the School of Engineering and Computer Science, Washington State University Vancouver, Vancouver, WA 98686, USA. E-mail: xuechen.zhang@wsu.edu*
- *Z. Li is with the School of Computer Science, Wuhan University, Luojiashan, Wuhan, Hubei 430072, China. E-mail: zongpeng@whu.edu.cn*

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

replica for each identified data access pattern of the application. As a result, the logical data in the replica space are reorganized in the order with how they are accessed by applications. After determining the reorganization, in *the "layout-aware" (LA) replica placement policy*, the system stores the generated replicas with their optimized data layouts in the same parallel file system (PFS) on HServers, based on the results of quantitative analysis on data access time cost. Once the replicas are ready, the I/O system is able to serve future data requests on the replicas with friendly access patterns.

Moreover, since replicating data across multiple HServers may still be insufficient for applications with high I/O concurrency, we propose an SSD-based PDLA replication scheme, S-PDLA, which utilizes SSD-based servers (SServers) to store replica files for improved I/O performance. As SServers do not always perform better than HServers for all access patterns and all servers working together may outperform SServers alone for some access patterns, S-PDLA selects critical patterns that can bring performance benefits to carry out cost-effective data replication on SServers or HServers+SServers. By leveraging the higher I/O performance of SSDs and the selective data replication policy, S-PDLA can both efficiently utilize SSD space and maximize I/O system performance for applications with a large number of processes.

Specifically, this study makes the following contributions.

- We propose an HDD-based PDLA data replication scheme, H-PDLA, which discovers an application's access patterns and places the replicas on multiple HServers with optimized data layouts based on access cost analyses.
- We propose an SSD-based PDLA data replication scheme, S-PDLA, which identifies the critical access patterns and selectively stores their replicas on SServers or all HServers+SServers with optimized layouts, to further improve I/O system performance.
- We implement both H-PDLA and S-PDLA within MPICH2 on top of OrangeFS. Experimental results with representative benchmarks and a real application show that H-PDLA can significantly improve parallel I/O system performance and demonstrates the performance benefit of S-PDLA over H-PDLA.

The rest of this paper is organized as follows. Section 2 gives the motivation of H-PDLA and S-PDLA. Section 3 and 4 describe the design and implementation of H-PDLA and S-PDLA respectively. Section 5 presents the evaluation results. Section 6 introduces the related work. Finally, Section 7 concludes this paper.

2 MOTIVATION

2.1 Motivation of H-PDLA

Applications exhibit I/O access patterns: Many HPC applications read or write data in certain ways. In other words, their I/O behaviors exhibit access patterns. For example, due to the iterative loop structures of program codes, some individual or group of data accesses may repeat with certain request distances for many times [10]. As these HPC applications often run multiple times to generate and analyze data, the same patterns can be identified in the same execution environment and configuration among different runs [11], [12]. Therefore, it is feasible to collect and utilize this information to guide data replication.

Contiguous access is preferable: Traditional HDDs are the dominant storage media in modern parallel I/O systems. As non-contiguous data accesses usually involve time-consuming disk head seeks, it brings higher I/O performance than that of a non-contiguous data request. Moreover, while SSDs have different storage characteristics compared to HDDs, they also exhibit better performance for contiguous I/O requests [13]. Based on this observation, it is preferable to generate more contiguous I/O requests through data replication to enhance I/O system performance.

Data layout affects I/O performance: Parallel file systems support multiple data layouts, which determine how the file data is distributed across multiple file servers. Typical approaches can distribute data on one server, a set of servers, and all servers. As shown in the previous work [8], [14], [15], data layout in parallel file systems can significantly influence I/O system performance because it involves different network and storage activities for given I/O requests. For applications with different data access behaviors, the optimized data layouts are different. To maximize the I/O system performance, an idea data placement scheme should choose the optimized data layout according to application data access characteristics.

2.2 Motivation of S-PDLA

H-PDLA is only suitable for applications with relatively low I/O concurrency. This is because the optimized data layout can perform well only when each server handles a small number of processes. However, in a large-scale HPC system, when the number of processes is significantly larger than the number of HServers, each HServer still needs to frequently switch among multiple processes, leading to very low I/O efficiency. Since flash-based SSDs have higher I/O performance and are much less sensitive to random accesses than HDDs, it is natural to propose S-PDLA, which adds additional SServers in the system to optimize data replication. However, it is non-trivial to do this due to the following observations.

First, replicating the data of all access patterns on SServers cannot always improve I/O system performance. Although an SSD usually outperforms an HDD, whether SServers outperforms the original HServers becomes uncertain. This is because the aggregated I/O bandwidth relies on multiple factors, such as the number of servers, the data layout, and the data access pattern [8], [16]. Therefore, S-PDLA needs to perform selective pattern replication to ensure performance benefits.

Second, all servers (HServers+SServers) working together may provide better I/O performance than HServers or SServers alone. Previous work has shown that, if with an optimized data layout, all servers may increase I/O performance for some access patterns (e.g., large requests) due to their higher I/O parallelism [17]. Therefore, although storing replica data on SServers is natural, an al-

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

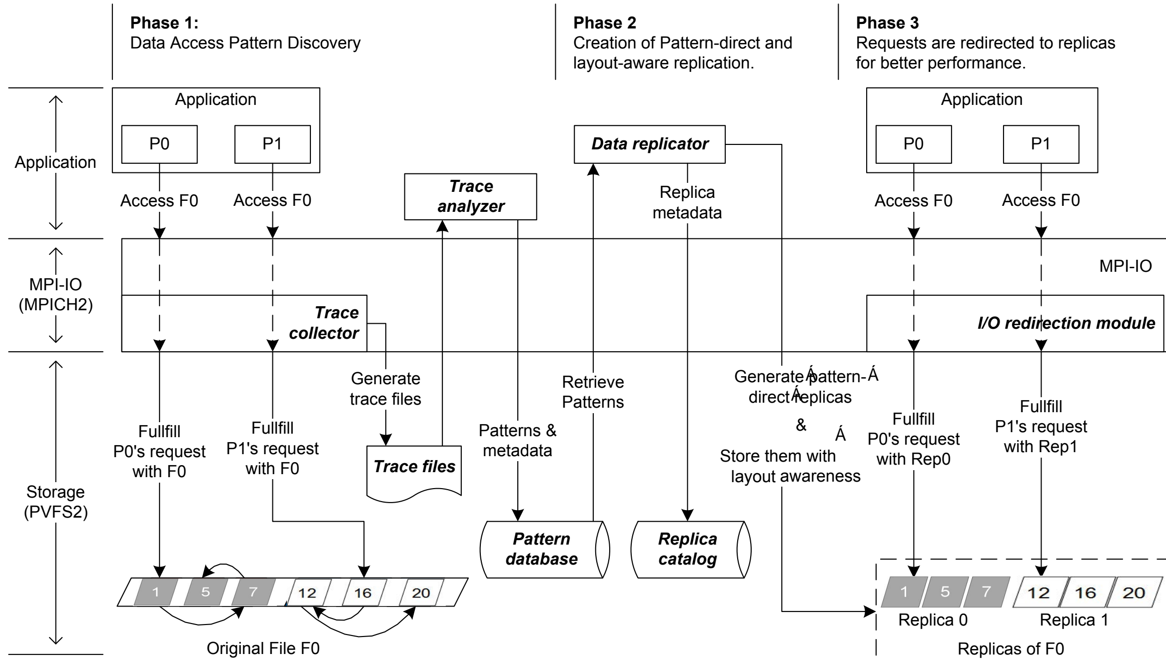


Fig. 1: The architecture of the H-PDLA data replication scheme.

ternative approach for S-PDLA is to store them on all HServers+SServers when doing so is beneficial.

3 The H-PDLA Design

3.1 Overview of H-PDLA

Figure 1 shows the system overview of the H-PDLA replication scheme, which resides in MPI-I/O library and consists of three phases in chronological order. In the first phase, during the application's first-time execution, the pattern recognition module identifies and saves the data access patterns. In the second phase, the system creates replicas directly according to the recognized data access patterns. Compared to the original data files, the replica files represent the data in the order of the data are accessed. In the third phase, the system automatically forwards I/O requests in the later runs of the same application to the replica files for better performance.

3.2 Data Access Pattern Identification

3.2.1 Trace Collector

This module is responsible for capturing MPI-IO calls in the MPI standard. It is an I/O middleware library, which can be linked to any application we want to trace. Other than this linking step, there is no need for program modifications. During execution, the application is linked to this I/O library and each process generates one trace file containing all its I/O operations. For each file operation, the trace collector gathers the following information: 1) the MPI rank and process ID; 2) the file ID; 3) the file offset and request size; 4) the name of the I/O operations; 5) the starting time of the operation; and 6) the ending time of the operation. The trace collector also records the mapping between the unique file ID and the file path.

3.2.2 Access Pattern Definition

Data accesses of many applications follow certain patterns. In this study, we define the I/O access pattern from five aspects: spatial locality, request size, temporal information, iterative behavior, and I/O operations. The spatial locality represents the byte order distance between successive accesses. They can be contiguous or non-contiguous or their combinations. The request size can be small, medium, or large. Small I/O requests commonly lead to I/O bottlenecks when they are sent to disks. The temporal information is classified based on intervals between data accesses, which can be fixed (the I/O occurs periodically) or random. The iterative behavior refers to the repetitive times of I/O requests, which can be repetitive or non-repetitive (i.e., only once). The I/O operation is divided into read-only, write-only, and read and write.

We categorize the data access patterns into two types: *local* and *global* ones. The local access pattern represents the I/O behavior of a process, which is a part of an application. However, in some situations, the local pattern cannot reflect the true story of the entire application. For example, when an application conducts collective I/O operations in which all the processes participate, the operations of each process are no longer separated behaviors. One can acquire the global access patterns of the application by co-analyzing the local I/O access patterns. More details about the definition of access patterns can be found in our previous work [11], [12].

While the pattern may include other properties, in this study we only focus on spatial patterns since they can substantially impact I/O performance [12], [5], [6]. We leave replication optimizations based on other patterns as the future work.

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

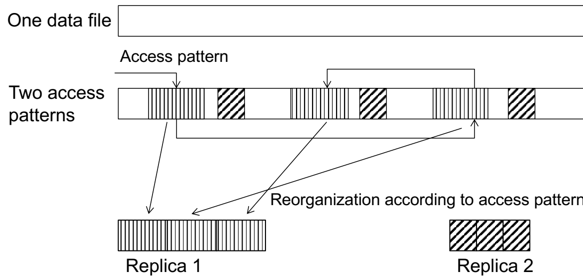


Fig. 2: Two access patterns on a parallel file. H-PDLA creates one replica for each pattern.

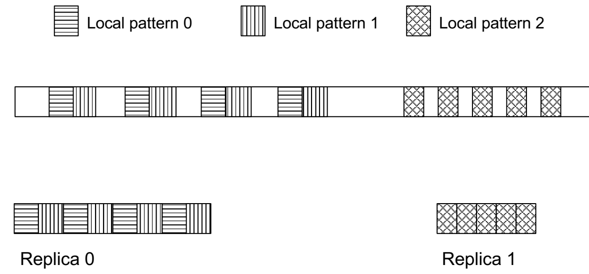


Fig. 3: Local patterns are combined into global patterns.

3.2.3 Access Pattern Collection

The trace analyzer identifies the data access patterns. It is an off-line procedure. By taking the collected trace files as inputs, the analyzer performs analysis to obtain the data access patterns. The detailed implementation of this module is described in Section 3.6. Once the data access patterns are obtained, the system stores them in a pattern database to guide the following data replication operations.

3.3 Pattern-Directed Replication Policy

3.3.1 Replica Creation Policy

The proposed scheme creates a replica for each identified access pattern besides the original data. Each replica contains a *data object* instead of the entire file. It only contains the data accessed in the data access pattern, as shown in Figure 2. The data that are not accessed or do not fall into any data access pattern, will not be replicated and only exist in the original file. Furthermore, the data in a replica are reorganized in the access order according to a corresponding access pattern. Each replica is stored as a new file in the same file system. Comparing to the trivial data replication, this policy yields more efficient utilization of storage space.

Generally, the number of replicas plays a pivotal role in data availability and performance. While distributed file system HDFS sets this parameter to three by default [18], we create one replica for each identified access pattern in parallel file system due to the following reasons. First, one replica provides comparable performance as $n(n > 1)$ replicas in HPC domains, where multiple processes seldom concurrently read the same data. Even when that happens, since application usually adopts collective I/Os to access data, there are only fewer processes to read data from storage nodes. Second, this policy is more efficient for storage space consumption. Third, it is simple to maintain data consistency between the replicas and the original data.

Since an application may have both local and global patterns, the scheme first creates replicas for all global patterns and then makes replicas for the local ones. This can reduce the number of data replicas and retain the flexibility of data layout optimization. As illustrated in Figure 3, local patterns 0 and 1 are combined, thus their data are in the same file–Replica 0. Local pattern 2 does not belong to any global pattern, and its data form an independent replica–Replica 1.

3.3.2 When to Create Replicas

The replica creation is an offline procedure. Namely, the proposed scheme does not create the replicas simultaneously with the running application. During the first execution of the application, the scheme discovers all data access patterns and then adds them into the I/O system's pattern database. Afterwards, the system starts to make replicas based on the queue of newly added patterns. These operations may be carried out only when there are free computing resources and I/O bandwidth, so that they do not affect the execution of normal tasks. In our future work, the system may allow users to add customized patterns into the queue. In this case, the future runs of the application will access other data with the specified patterns, so that the system can obtain improved I/O performance.

3.3.3 Where to Store Replicas

We store the replica files in the same parallel file system where the application runs. The replica files are only visible to the I/O middleware that redirects application I/O requests from the original files to the replicas. All replicas are placed in some specially named directories, and any naming rule would work as long as the system's meta-data keeps the replicas' file paths. Since modern parallel file systems support file data layout adjustment by setting the attributes of the file directories, storing the replica files in separated directories brings convenience for the layout-aware replica placement.

Like almost all other replication schemes, the pattern-directed replication strategy consumes additional storage space. It is a trade-off between data access performance and storage capacity. But, for many applications this is a good trade-off from the energy saving point-of-view. Reducing data access time will reduce energy consumption. In addition, since the replicas are a small portion active data of the original data, the original data then could be stored on slow spin disks or even on tapes. For this kind of applications, the trade-off of space becomes blurry and the gains in I/O performance and energy consumption become obvious. We will not explore energy saving in this study, but focus on I/O optimization.

3.4 Layout-Aware Replica Placement Policy

3.4.1 Data Layout in Parallel File Systems

In this study we categorize popular data layout schemes into three types: one-dimensional horizontal (1-DH), one-dimensional vertical (1-DV), and two-dimensional (2-D) data layouts. As shown in Figure 4, 1-DH places the data of

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

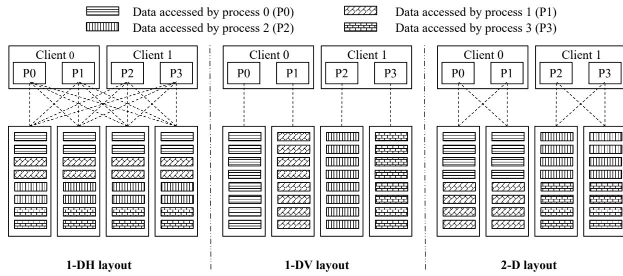


Fig. 4: Three typical data layouts in parallel file systems.

TABLE 1: Parameters in the data access cost model.

Symbol	Description
p	Number of I/O client processes
m	Number of processes on one client node
s	Data size of one access
e	Time of single network connection establishing
v	Network transmission rate
n_h	Number of HDD servers for data replication
a_h	Startup time of one I/O operation on HDD
b_h	Time of reading/writing one unit of data on HDD
g_h	Number of storage groups in 2-D layout

each process across all available storage nodes. While 1-DV distributes the data of each process on one storage node, 2-D distributes each process's data on a group of storage nodes.

Among these schemes, 1-DH is the most widely used one because it can provide decent I/O performance in many situations. It is the default data layout method namely "simple striping" in OrangeFS. However, in some cases, it yields poor performance. For example, when the number of processes is much larger than the number of storage nodes, each I/O request will suffer a large latency because each storage node needs to serve concurrent requests from all processes [4]. In this case, 1-DV provides higher I/O performance than 1-DH does because each node needs to serve fewer processes. This example shows that for different data access patterns, we should choose different data layout to yield the best I/O system performance.

3.4.2 Cost Model for HDD-Based Servers

Replicas created by the pattern-directed replication scheme are logical files before getting stored. While storing them into parallel file systems, the layout-aware data storage optimization first needs to identify the optimized data layouts for them. For this purpose, we leverage a mathematical model to evaluate the data access cost in the computing environment. The corresponding critical parameters are listed in Table 1. Assume T_H^{1V}, T_H^{1H}, and T_H^{2D} represent the data access times in the HDD-based file system under 1-DV, 1-DH, and 2-D layouts respectively, then the cost model is listed in Table 2. This model is effective enough and has been comprehensively verified in a practical parallel I/O system. More details about constructing and verifying the model can be found in our previous research [8].

We can derive some brief guidelines can from the model. 1) When $p < n$ and s is very large, 1-DH policy has the lowest cost among the three policies. 2) When $p \approx n$ and s is medium, 2-D layout policy produces higher bandwidth than the other two. 3) When $p > n$ and s is relatively small, 1-DV layout policy would be the best choice.

TABLE 2: Cost model for HDD-based servers.

Cost	Description
T_H^{1V}	$max(m, \lceil \frac{p}{n_h} \rceil) * (e + sv) + \lceil \frac{p}{n_h} \rceil * (a_h + sb_h)$
T_H^{1H}	$max(p, mn_h) * e + max(\frac{p}{n_h}, m) * sv + pa_h + \frac{p}{n_h} * sb_h$
T_H^{2D}	$max(m\lceil \frac{p}{n_h} \rceil, \lceil \frac{p}{g_h} \rceil) * e + max(m, \frac{\lceil \frac{p}{g_h} \rceil}{\lceil \frac{n_h}{g_h} \rceil}) * sv + a_h \lceil \frac{p}{g_h} \rceil + \frac{\lceil \frac{p}{g_h} \rceil}{\lceil \frac{n_h}{g_h} \rceil} * sb_h$

3.4.3 Layout-Aware Replica Placement

Based on the proposed cost model, the *layout-aware* replica placement scheme works as follows. First, for each request in a given access pattern, the cost model estimates its access cost under each of the three data layouts. Then it adds all the requests' costs together to get the access cost for the entire pattern. Also, for each data layout, the model generates a cost result. Naturally, the data layout that produces the lowest cost is the optimized selection, and the scheme will adopt this optimized layout in the parallel file system for the corresponding replica placement.

3.5 I/O Redirection

The I/O redirection module redirects original I/O requests to the newly created replicas. Usually an application issues an I/O request with three parameters: the identifier of the original file, the data offset, and the request size. After locating the replica file according to these three parameters, run-time information (the parameters listed in Table 1), and the meta-data in the replica catalog, the redirection module translates the filename and offset between the original file and the replica files and fulfills the request using the replica files.

3.6 Implementation

We implement a prototype of the H-PDLA replication scheme and its run-time system under MPICH2 on top of OrangeFS. The implementation adds some components into the default parallel I/O system. We explain the detail of each component hereinafter.

3.6.1 Trace Collector and Analyzer

While there are several tools that can be used to collect traces and analyze patterns, we choose IOSIG [11] because it can capture the required information of our scheme and only incurs low overheads. IOSIG is a pluggable library at MPI-IO layer. By using the Profiling MPI interface, it collects the data access information of all file I/O operations. All the file operations of an application are recorded in trace files, each for one individual process. Besides the MPI-IO interface, IOSIG supports standard POSIX IO interfaces for portable deployment.

The trace analyzer is implemented in Python to carry out offline trace analysis. It utilizes a "template matching"

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

approach to recognize data access patterns, which are represented in the form of *I/O signatures* defined in our previous work [12]. By marking the analysis progress with a cursor in each trace file, the analyzer starts from the first record and moves the cursor forward to scan all records until reaching the end. During the scanning process, the analyzer always picks a predefined access pattern as a template, to check whether it matches the records around the cursor. Once a match is found, the cursor moves forward along with the same pattern in the trace, until the match does not hold. If there is no match for the first template, the analyzer switches to other templates and tries again. If failing to find a match for all templates, it skips the current record, moves the cursor forward, and starts over the matching at the new position. The analyzer produces all the local patterns by analyzing each trace file. After that, it examines all the local patterns and combines the relative ones into global patterns. Trace analyzer inserts obtained patterns into Pattern Database.

3.6.2 Pattern Database

Both the data replicator and the I/O redirection module need to retrieve an application's data access patterns. We keep this meta-data information in "Pattern Database". It saves the mapping relation between applications and their data access patterns, including: 1) which application a pattern belongs to; 2) which file a pattern depends on; 3) the rank of a process that a pattern belongs to; and 4) which local access pattern is included in a global access pattern. Pattern Database also saves the meta-data on the run-time environment of the owner application, including mainly the parameters of the system that will be used to determine the optimized data layout for the generated replication files.

We use *Berkeley DB* to implement the pattern database. The *Berkeley DB* is a hash table and each of its records is a key-value pair. The key is a "patternID" that is encoded by the following information: application's execution command, number of processes, rank of the process, and the original file name; the value contains the data access pattern and the run-time information. The value's presentation in the code is a structure definition (in C language) that includes several fixed member variables and a union (also in C language) of various type of data access patterns.

3.6.3 Data Replicator

The data replicator is a lightweight daemon program that monitors a queue containing all data access patterns needed to be replicated. When the trace analyzer inserts a new access pattern into the pattern database, it also en-queues the same pattern into this global queue. When the queue is not empty, the replicator will de-queue the access patterns and starts to make replication according to them, one at a time. In the meantime, the data replicator uses the run-time information (the parameters listed in Table 1) and the cost model to find the optimized data layout. Then it just reads data from the original file and writes them into the replica files with the optimized data layout. We implement such a queue using *Berkeley DB*'s built-in queue access mode. To configure the data layout, the data replicator just sets up a directory with the optimized data layout in OrangeFS, and then stores the replicas into that directory. OrangeFS provides a tool for configuring a directory's data layout policy.

The data replicator also works like a replica scanner. It periodically scans the pattern database, and whenever it finds the missing of a pattern's original file, it will remove the corresponding data replicas and the related meta-data.

3.6.4 Replica Catalog

The replica catalog stores the meta-data of the replicas. It manages the relationships among data replicas, original files, and the data access patterns. The meta-data includes 1) which original file a replica's data comes from, and 2) based on which access pattern a replica is created. Its implementation also uses *Berkeley DB* configured as a hash table; the key is the patternID (the same key in Pattern Database), and the value is the path to the replica file based on the corresponding data access pattern.

3.6.5 I/O Redirection

This module redirects I/O requests to the original files or the replica files based on the specifics of the requests. Usually, an application issues an I/O request with three parameters: the identifier of the original file, the data offset, and the request size. After locating the replica file according to these three parameters, run-time information, and the meta-data in the replica catalog, the redirection module translates the filename and offset between the original file and the replica and fulfills the request using the replica.

We have made the following modifications in the MPI-IO standard functions to redirect I/O requests based on request specifics.

`MPI_File_open`: When opening a file, instead of opening the original file, the method tries to open the corresponding replica files.

`MPI_File_read/MPI_File_write` (and other formats of read/write, such as `MPI_File_read_at`, etc.): For each I/O read or write, this function uses the file handle of the replica file and checks whether the access pattern has changed or whether the opened file contains the requested content. If the application is still following the same pattern, the module calculates the correct data offset and issues a request using the new offset and file handle. If the pattern has changed, the module finds new patterns, opens new replication files, and issues request to them.

`MPI_File_close`: It closes the opened replica files.

`MPI_File_seek`: It calculates the offset and conducts the seek operation in the replica if necessary.

When the requested data do not belong to any data access pattern and do not have replicas, the system will act as the same as the default MPI-IO implementation.

4 The S-PDLA Design

In the previous section, we have described the H-PDLA replication scheme for HDD-based parallel I/O systems. As discussed in Section 2.2, H-PDLA is inefficient for an application with high I/O concurrency. To address this issue, we propose a new SSD-based replication scheme, S-PDLA to further improve I/O system performance.

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

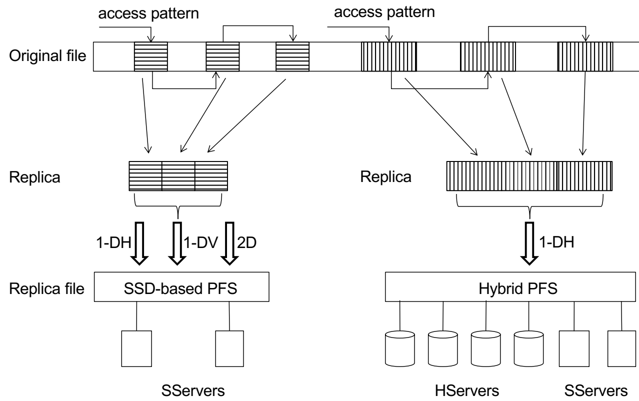


Fig. 5: The idea of the S-PDLA scheme. The data of access patterns can be replicated on SServers or HServers+SServers based on the specifics of the patterns.

4.1 Idea of S-PDLA

The basic idea of S-PDLA is to utilize SServers as the replica space for the original file system. Since (1) SServers do not always perform better than HServers for all access patterns and (2) all servers may outperform the pure SServers for some access patterns, S-PDLA only selects critical patterns that can bring significant performance benefits to carry out cost-effective data replication on SServers or HServers+SServers.

Figure 5 shows the idea of the S-PDLA scheme. It replicates some of the identified access patterns on SServers and others on HServers+SServers when the data replicas on the original HServers are inefficient. Where to store the replica of the pattern is determined by the replication benefit analysis. During the subsequent I/O service phase, S-PDLA adaptively redirects I/O requests to SServers or HServers+SServers based on the specifics of the requests. In contrast to the simple scheme that indiscriminately replicates the data of all access patterns on SServers, S-PDLA makes selective data replication to both save SSD space and maximize I/O system performance. S-PDLA distributes the replica data on SServers with optimized data layouts since they also affect SSD-based I/O performance [1], [4]. For simplicity, S-PDLA only stores replica data on HServers+SServers with the 1-DH layout. The other two types of data layouts can also be applied to S-PDLA [4], [9]; we will integrate them into S-PDLA in the future work.

4.2 Cost Models for SServers and HServers+SServers

To enable the cost-effective data replication, we propose two cost models to evaluate the data access time of a file request on different types of servers.

4.2.1 Cost Model for SServers

For SServers, the related parameters in the model are listed in Table 3 and the cost model is defined in Table 4. While similar to the previous model in Section 3.4.2, the model shows three differences. First, the number of SServers can be different from HServers in the HDD-based model. Second, the read and write performance of SServers are different because a write on SSD usually involves background activities (e.g., garbage collection and wear leveling) while a read does not. Third, the startup time and unit data access time of SSDs are much smaller than those of HDDs because SSDs have much higher I/O performance. By considering these configuration and device differences, the model can efficiently evaluate request performance on SServers

TABLE 3: Parameters in cost model for SServers.

Symbol	Description
p	Number of I/O client processes
m	Number of processes on one I/O client node
s	Data size of one access
e	Time of single network connection establishing
v	Network transmission rate
n_s	Number of SSD servers for data replication
a_{sr}	Startup time of one read operation on SSD
b_{sr}	Time of reading one unit of data on SSD
a_{sw}	Startup time of one write I/O operation on SSD
b_{sw}	Time of writing one unit of data on SSD
g_s	Number of storage groups in 2-D layout

TABLE 4: Cost model for SSD-based servers.

Cost	Description
	Read operation
T_S^{1V}	$max(m, \lceil\frac{p}{n_s}\rceil)*(e+sv)+\lceil\frac{p}{n_s}\rceil*(a_{sr}+sb_{sr})$
T_S^{1H}	$max(p, mn_s)*e+max(\frac{p}{n_s}, m)*sv+pa_{sr}+\frac{p}{n_s}*sb_{sr}$
T_S^{2D}	$max(m\lceil\frac{p}{n_s}\rceil, \lceil\frac{p}{g_s}\rceil)*e+max(m, \frac{\lceil\frac{p}{g_s}\rceil}{\lceil\frac{n_s}{g_s}\rceil})*sv+a_{sr}\lceil\frac{p}{g_s}\rceil+\frac{\lceil\frac{p}{g_s}\rceil}{\lceil\frac{n_s}{g_s}\rceil}*sb_{sr}$
	Write operation
T_S^{1V}	$max(m, \lceil\frac{p}{n_s}\rceil)*(e+sv)+\lceil\frac{p}{n_s}\rceil*(a_{sw}+sb_{sw})$
T_S^{1H}	$max(p, mn_s)*e+max(\frac{p}{n_s}, m)*sv+pa_{sw}+\frac{p}{n_s}*sb_{sw}$
T_S^{2D}	$max(m\lceil\frac{p}{n_s}\rceil, \lceil\frac{p}{g_s}\rceil)*e+max(m, \frac{\lceil\frac{p}{g_s}\rceil}{\lceil\frac{n_s}{g_s}\rceil})*sv+a_{sw}\lceil\frac{p}{g_s}\rceil+\frac{\lceil\frac{p}{g_s}\rceil}{\lceil\frac{n_s}{g_s}\rceil}*sb_{sw}$

4.2.2 Cost Model for HServers+SServers

For simplicity, we only consider the 1-DH data layout. Since HServers and SServers have heterogeneous performance, we assume the data are distributed on HServers and SServers with a varied-size file stripe size of s_h and s_s respectively to alleviate load imbalance among servers. We assume each file request is served by all the n_h+n_s servers, namely, $n_h*s_h+n_s*s_s=s$. Usually s_s is larger than s_h to achieve load balance. In an extreme case, s_h can be zero (which means file data are only distributed on SServers) if there is a possibility to improve performance. Based on these assumptions, the corresponding cost model is listed in Table 5.

Note that the data access time in the model is a function of two variables, s_h and s_s for a given file request. It consists of linear equalities and inequalities of unknown variables (the max expression can be described as multiple linear inequalities). Therefore, we can use a linear programming (LP) optimization technique or an iterative algorithm to find the optimized stripe sizes with the given assumptions. Since the linear program is expressed with only two unknown

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

TABLE 5: Cost model for HServers+SServers.

Cost	Description
	Read operation
T_{H+S}^{1H}	$max(p, m(n_h + n_s)) * e + max(ms, ps_s) * sv + p * \max(a_h + s_h * b_h, a_{sr} + s_s * sb_{sr})$
	Write operation
T_{H+S}^{1H}	$max(p, m(n_h + n_s)) * e + max(ms, ps_s) * v + p * \max(a_h + s_h * b_h, a_{sw} + s_s * sb_{sw})$

variables, the search space is very small and solving the program requires acceptable off-line time overhead. More details on constructing the model and finding the optimized stripe sizes for a given access can be found in [4].

4.3 Critical Access Pattern Identification

With the proposed cost models, S-PDLA can get the data access costs of each request r_i on HServers, SServers, or HServers+SServers under different data layouts. For a given access pattern, assume that n is the number of request in the pattern, we can obtain the optimized data access time for the pattern on the three potential replica spaces as following:

$$T_H = \min\{\sum_{i=1}^{n} T_H^{1H}(r_i), \sum_{i=1}^{n} T_H^{1V}(r_i), \sum_{i=1}^{n} T_H^{2D}(r_i)\} \quad (1)$$

$$T_S = \min\{\sum_{i=1}^{n} T_S^{1H}(r_i), \sum_{i=1}^{n} T_S^{1V}(r_i), \sum_{i=1}^{n} T_S^{2D}(r_i)\} \quad (2)$$

$$T_{H+S} = \sum_{i=1}^{n} T_{H+S}^{1H}(r_i) \quad (3)$$

Among the optimized data layouts, the optimal one yields the lowest data access costs on the three replica spaces. For a given data access pattern, the performance benefits of an intended replication on SServers over HServers can be described as

$$B_S = T_H - T_S \quad (4)$$

Accordingly the performance benefits of an intended replication on HServers+SServers instead of HServers can be described as

$$B_{H+S} = T_H - T_{H+S} \quad (5)$$

Assume the maximal performance benefit of req is B, then B can be described as

$$B = \max\{B_S, B_{H+S}\} \quad (6)$$

A positive B means that the intended data replication on SServers or HServers+SServers will further improve I/O system performance than replicating the pattern data on HServers. The larger the value of B, the more benefit the replication brings. However, since SServers do not always outperform HServers for all system configurations and access patterns, the intended replication is not always profitable. For example, when the system has a large number of HServers and a small number of SServers, HServers may have better aggregated I/O bandwidth than SServers because they provide higher I/O parallelism. In this case, serving the required data from the replicas on HServers (original I/O system) helps system performance and it does not need to be replicated on SServers. We will show this in Section 5.2.

To maximize system performance, S-PDLA regards the access patterns whose values of B are larger than a predefined threshold as critical patterns and then replicates them on SServers or HServers+SServers. Where to store the replica is determined by the replica space which brings the maximal performance benefit. To guide the following actual data replication, S-PDLA uses a critical pattern table (CPT) to store the pattern information, the performance benefit, the replica space, and the corresponding optimized data layouts.

4.4 Replica Creation and Placement for Critical Patterns

Algorithm 1 The SSD-based replica creation and placement

Require: Critical pattern table: CPT, Replica Mapping table: RMT.
1: $pattern[n] \leftarrow sort_desc_benefit(CPT[n])$
2: **for** $(i = 0; i < n; i++)$ **do**
3: $opt_layout \leftarrow pattern[i].layout$
4: $c_{sys} \leftarrow$ available space on an SServer
5: $s_{pat} \leftarrow$ required data size of $pattern[i]$ on an SServer with opt_layout
6: **if** $c_{sys} >= s_{pat}$ **then**
7: **if** pattern[n].space ==”SServers” **then**
8: create a file $f[i]$ with opt_layout on SServers
9: **else**
10: create a file $f[i]$ with opt_layout on HServers+SServers
11: **end if**
12: **for** each request $r[j]$ in $pattern[i]$ **do**
13: **if** $r[j]$ is $\notin f[i]$ **then**
14: append the data of request $r[j]$ to file $f[i]$
15: update the entry into RMT
16: **end if**
17: **end for**
18: **end if**
19: $c_{sys} \leftarrow c_{sys} - s_{pat}$
20: **end for**

Once obtaining the critical access patterns, S-PDLA will replicate them on SServers or HServers+SServers with optimized data layouts. Algorithm 1 shows the replica creation and placement procedure. To be cost-effective, it first sorts all the critical patterns in descending order of their performance benefits, then iterates them for possible data replication from high-benefit patterns to low-cost ones. A pattern with a higher benefit has a more top priority to be replicated. For each access pattern, the algorithm checks whether the system has enough space to accommodate the involved data with the optimized data layout. If it is yes, the algorithm will create a replica file for the given pattern in the corresponding replica space (SServers or HServers+SServers). Then it continuously copies the requested data from the

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

original file to the replica file according to their accessing order in the original file. To keep up replica location and layout information, the algorithm adds an entry in the replica mapping table RMT when new data are appended into the replica file. Otherwise, the access pattern will be not replicated due to the insufficient space.

Note that the algorithm is a selective policy that only replicates the critical access patterns instead of all of them. Furthermore, it is layout-aware as it stores each replica on SServers or HServers+SServers with an optimized data layout rather than the fixed *1-DH* data layout. In this way, the limited space of SServers can be efficiently utilized.

4.5 Implementation

CPT and RMT are two key data structures of S-PDLA to record critical patterns and replica mapping information. We use *Berkeley DB* [19] to implement both of them, each as a database file in the parallel file system on SServers. For *CPT*, the key is the PatternID encoded with the application name, number of processes, rank of the process, original file name, and the access pattern; the value contains the performance gain, the replica space, and the optimized data layout. For *RMT*, the key is the PatternID and the value contains the data access information. Additionally, we use lists in memory to maintain the most frequently accessed entries in the two tables to speed up lookups.

We modify the MPI library so that the replica location information on SServers or HServers+SServers is loaded with `MPI_Init()` and unloaded with `MPI_Finalize()`. We also modify the `MPI_File_open/read/write/seek/close()` functions (and other variants of them) to atomically forward I/O requests to the corresponding replica spaces.

5 Evaluation

We conduct the experiments on a 64-node Sun Fire Linux cluster. Each node has two Opteron quad-core processors, 8GB memory and a 250GB 7200RPM SATA-II disk (HDD). We employ eight nodes as storage nodes managed by OrangeFS and all the other nodes as computing nodes. There was no overlap between computing nodes and I/O nodes (file servers), so that all data accesses between the application and the file system are remote accesses.

We use three popular micro-benchmarks, IOR [20], PIO-Bench [21], MPI-Tile-IO [22], and a real application [23] to evaluate the system performance. We test the system performance five times and chose the averages as the results.

5.1 Evaluation on the H-PDLA scheme

The evaluation in this subsection is to show the effectiveness of the HDD-based pattern-directed (PD) and layout-aware (LA) data replication scheme. For comparison, we test the performance of the original HDD-based parallel I/O system (Original) without data replication. To ensure that the improvements only come from applying the PD replication policy, we first disable the LA placement in these tests. Therefore, the replicas and the original files are using the same data layout. The data layout is 1-DH (”simple striping” distribution in OrangeFS) with a stripe size of 64KB.

Then, to verify the need of optimized data placement of the replicas, we measure the system performance by enabling both PD and LA (PD+LA, namely H-PDLA). We store the replicas in two different ways, one set of them are stored in OrangeFS's default data layout (1-DH), and the other set of replicas are stored in the optimized data layout determined by the replica placement scheme in Section 3.4. These two sets of replicas are identical with each other, so all the performance differences are the result from the differences in their data layouts.

5.1.1 IOR Benchmark

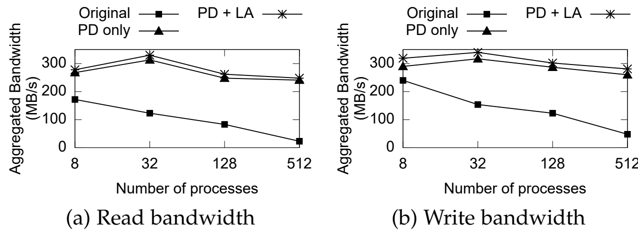
Various Numbers of Processes: Figure 6 shows the I/O performance of IOR with varying number of processes. We run IOR with 8, 32, 128, and 512 processes. Each process accesses 512MB of data in a fixed-stride data access pattern, and the request size is 256KB. Different processes access their regions of the original file so that no process's data co-locate with any other's data. We can observe that both PD and H-PDLA can improve the performance of the original I/O system. Compared to the original system, the PD policy improves read performance by up to 970% and write performance by up to 307%. The reason is that in the original I/O system, there are many non-contiguous I/O requests, which degrade I/O efficiency. However, with the PD policy, these requests are translated into sequential requests in the replica space, leading to highly improved I/O bandwidth. In contrast to the PD policy, the LA replication placement policy can produce up to 10% additional performance improvements for reads and 5% for writes. This is because PD only applies one fixed data layout for replica placement while LA can choose the best one requiring the lowest cost from three layout candidates. From these results, we can see that the optimized data placement for replicas can further improve I/O system performance.

We also note that as the number of processes increases, IOR's bandwidth gets lower because each storage node needs to serve more processes, leading to more severe contention on storage nodes. However, the rate of bandwidth degrading is much lower with both PD and H-PDLA than that with the original case. This shows that the proposed data replication scheme can significantly improve the system scalability on serving more concurrent requests.

Various Request Sizes: Figure 7 shows the bandwidths of IOR with various request sizes of 16KB, 256KB, and 4MB. We set the number of processes to 64. Similar to the previous test, both the PD and H-PDLA scheme can obtain performance improvements over the original I/O system. In contrast to the original I/O system without data replication, the PD replication improves the I/O bandwidth by 80% to 926% for reads, and 47% to 521% for writes. With the request size getting smaller, IOR's bandwidth gets lower. This is because each storage node needs to handle a larger number of small non-contiguous data requests thus the disk seeks get more frequent. Figure 7 also reveals another improvement by applying PD replication: when the request size decreases, the rate of bandwidth degrading is much lower with PD than that with the original case. In other words, PD can

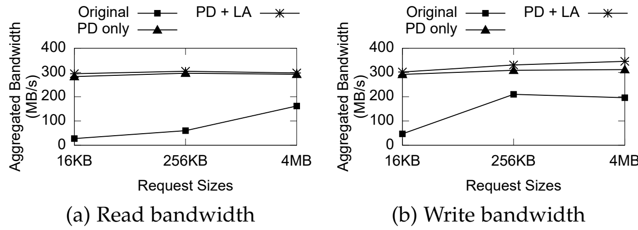
0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers



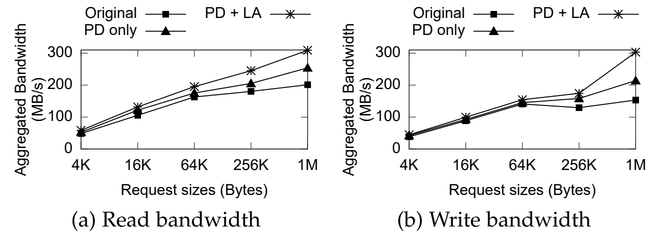
(a) Read bandwidth (b) Write bandwidth

Fig. 6: IOR performance with various numbers of processes.



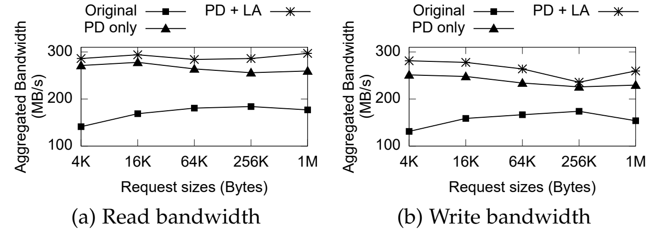
(a) Read bandwidth (b) Write bandwidth

Fig. 7: IOR performance with various request sizes.



(a) Read bandwidth (b) Write bandwidth

Fig. 8: PIO-Bench performance with various request sizes.



(a) Read bandwidth (b) Write bandwidth

Fig. 9: MPI-Tile-IO performance with various request sizes.

significantly improve the I/O system's ability to handle a large number of small requests.

When the LA placement policy is enabled, Figure 7 shows that it can further improve the I/O performance obtained by the PD replication policy, meaning that LA is useful and necessary. Compared to the original I/O system, the I/O system performance can be improved by 84% to 970% for reads and 16% to 208% for writes, when both the PD and LA policy are enabled.

5.1.2 *PIO-Bench Benchmark*

The PIO-Bench benchmark can test several I/O access patterns appearing in typical workloads of real applications. These patterns include sequential, simple strided, nested strided, random strided, segmented access, and tiled patterns. We run PIO-Bench with a nested-stride access pattern. The number of processes is 64, and the request sizes are 4KB, 16KB, 64KB, 256KB, and 1MB. We record the program's execution time and use it to divide the total data access size to get the aggregated I/O bandwidth.

Figure 8 shows the performance improvements that PD brings to PIO-Bench. Compared to the original I/O system, we can see that PD has performance improvement of 8% to 27% for reads, and 10% to 39% for writes. Figure 8 also reveals that the LA optimization can have extra performance benefits over the PD policy alone: the improvement is 9% to 22% for reads and 5% to 41% for writes. These results show that H-PDLA is effective for PIO-Bench.

As mentioned above, the data access patterns of PIO-Bench are nested-stride. This means each process has a fixed-stride access pattern. But multiple local access patterns are nested with each other and can be combined into global access patterns. Therefore, the nested-stride pattern yields better data locality than the fixed-stride data access pattern that we used for IOR's tests. As a result, the performance improvements of PIO-Bench are not as substantial as those of IOR but are still significant. This further confirms the adaptability of H-PDLA: when the application's data accesses have poorer performance (due to the worse data locality among consecutive accesses), it gains more benefits from H-PDLA.

5.1.3 *MPI-Tile-IO Benchmark*

We then run MPI-Tile-IO with its default access pattern. MPI-Tile-IO treats the entire data file as a 2-D matrix and divides it into $n \times n$ tiles (n rows by n columns). Given n^2 processes, each process accesses the data in one tile, with fixed-stride access pattern. Each process accesses a chunk of data based on the size of each tile and the size of each element. The data of n tiles in the same row are nested together. Therefore, MPI-Tile-IO's data access pattern is also nested-stride.

In this test, we run MPI-Tile-IO with 64 processes and various request sizes. The request sizes are 4KB, 16KB, 64KB, 256KB, and 1MB. Figure 9 shows the performance improvements that PD brings to MPI-Tile-IO over the original I/O system. The aggregated bandwidth increases by 39% to 91% for reads, and 30% to 86% for writes. Figure 9 also illustrates the performance improvements that H-PDLA brings to MPI-Tile-IO over the PD scheme: the aggregated I/O bandwidth is improved by 6% to 15% for read requests and 4% to 14% for write requests, meaning that the LA scheme is efficient for MPI-Tile-IO.

Similar to the PIO-Bench test, while the performance improvement of MPI-Tile-IO is significant, it is not as large as that of IOR. The reason is that the nested-stride pattern used in MPI-Tile-IO has strong data locality than the fixed-stride data access pattern used in IOR.

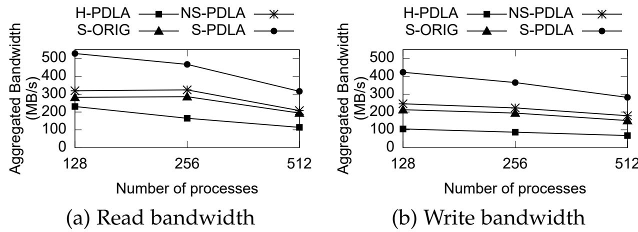
5.2 Evaluation on the S-PDLA Scheme

All the above results have confirmed the efficiency of H-PDLA in some cases. In this subsection, we conduct experiments to show that the enhanced S-PDLA scheme can further improve I/O performance for applications when the I/O concurrency is significantly larger than the number of servers.

We compare S-PDLA with three other schemes: *H-PDLA*, *S-ORIG*, and *NS-PDLA*. As previously discussed, H-PDLA

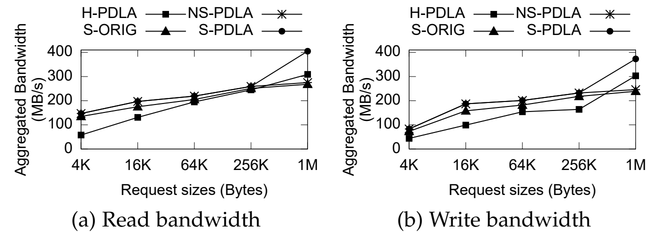
0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers



(a) Read bandwidth (b) Write bandwidth

Fig. 10: IOR performance with various process numbers.



(a) Read bandwidth (b) Write bandwidth

Fig. 11: PIO-Bench performance with various request sizes.

replicates all identified access patterns on HServers. S-ORIG stores the original files on SServers without PDLA. NS-PDLA is a non-selective policy, which indiscriminately replicates the data of all patterns on SServers without considering replication benefits. In contrast, S-PDLA only replicates the patterns that can bring performance benefits on SServers or HServers+SServers. With these comparisons, one will be clear about the necessity of the selective policy in S-PDLA and its efficiency in improving I/O system performance.

We deploy the original I/O system on eight HServers. Moreover, we build an additional system on four SServers, which are used to store the original files in S-ORIG and the replica files in S-PDLA.

5.2.1 IOR Benchmark

To simulate the scenario where each server faces high I/O concurrency, we run IOR with 128, 256, and 512 processes. We modified IOR to generate various request sizes in different parts of a file, so that processes access the data with varying I/O patterns. Each process accesses data with strided I/O requests. We vary request sizes among 16KB, 256KB, and 4MB in different parts of the file. For each process, the total accessed data size is 512MB.

Figure 10 shows the aggregated bandwidths of IOR under various numbers of processes. We find S-PDLA outperforms H-PDLA: the read performance is improved by 128% to 183%, and the write performance is improved by 252% to 279%. This is because S-PDLA leverages high-performance SSDs and optimized data layouts to enhance system performance. S-PDLA is also better than S-ORIG: it speeds up reads by up to 86% and writes by 98%. There are two reasons for the improvements. (1) S-PDLA stores the replicas with optimized data layouts while S-ORIG uses the default layout. (2) S-PDLA stores the replicas of some patterns on HServers+SServers while S-ORIG only stores them on SServers, which have lower I/O parallelism.

Another observation is that S-PDLA exceeds NS-PDLA because it carries out cost-effective replication while NS-PDLA performs indiscriminate data replication. To confirm this, we profile the request locations under S-PDLA. It shows that S-PDLA issues the requests whose sizes are smaller than 4MB to SServers and those at the size of 4MB to HServers+SServers. As an SServer is significantly faster than an HServer for small requests, S-PDLA replicates the patterns only on SServers is more beneficial because it can efficiently utilize SServers' performance. However, when the request size is 4MB, the performance gap between heterogeneous servers substantially decreases. In this case, HServers provide better I/O bandwidth than SServers because they have higher I/O parallelism. However, replicating the data on HServers+SServers can further improve I/O performance because it can maximally utilize the I/O parallelism from all nodes.

We also note that S-PDLA brings diminishing performance improvements as the number of processes increases. The reason is that SSDs also have decreased I/O performance when serving a large number of processes. However, as SSDs have higher I/O performance and less sensitivity to random accesses, S-PDLA can exhibit better performance and scalability than H-PDLA.

5.2.2 PIO-Bench Benchmark

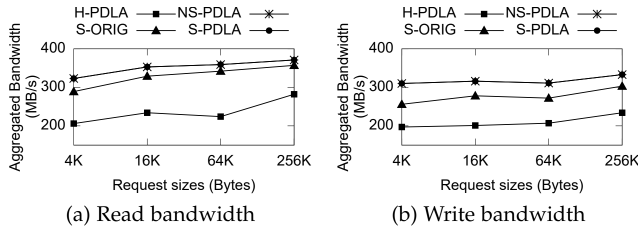
We run PIO-Bench with the nested-stride access pattern. The number of processes is 128 and the request size is from 4KB to 1MB. We calculate the aggregated I/O bandwidth by using the total data access size to divide the program's execution time. Figure 11 shows the I/O bandwidth of PIO-Bench with various request sizes. When the request size is equal or smaller than 256KB, S-PDLA obtains I/O performance improvement over H-PDLA by up to 153% for reads and by up to 89% for writes. In these cases, S-PDLA sends the I/O requests to SServers. When the request size is 1MB, S-PDLA obtains a much higher I/O bandwidth because in this scenario HServers outperform SServers and S-PDLA replicates the patterns on HServers+SServers. Similar to previous tests, S-PDLA is better than S-ORIG and NS-PDLA. The reason for S-PDLA's improvements is that it can intelligently replicate data of some patterns on SServers or HServers+SServers with optimized data layout, based on the replication benefit analysis. In contrast, S-ORIG only stores the data on SServers with the default layout, and NS-PDLA indiscriminately replicated the data on SServers even when HServers have superior I/O performance.

5.2.3 MPI-Tile-IO Benchmark

We run MPI-Tile-IO with the default access pattern. The number of processes is 256, and the request sizes are 4KB, 16KB, 64KB, and 256KB. Figure 12 shows the performance improvements that S-PDLA brings to MPI-Tile-IO compared with the H-PDLA, S-ORIG, and NS-PDLA replication scheme. We can see that S-PDLA can further increase the aggregated bandwidth by 21% to 60% for read requests and 31% to 57% for write requests compared to H-PDLA. Furthermore, since the request sizes are relatively small, S-PDLA obtains the same performance as NS-PDLA because both of them redirect the requests in all identified patterns to SServers.

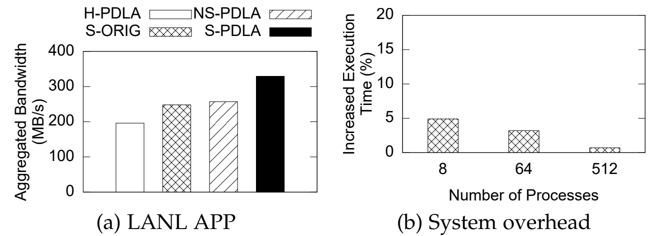
0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers



(a) Read bandwidth (b) Write bandwidth

Fig. 12: MPI-Tile-IO performance with different request sizes.



(a) LANL APP (b) System overhead

Fig. 13: Application performance and system overhead.

5.2.4 Real Application

The above benchmark-based evaluation has illustrated the effectiveness of S-PDLA. In this subsection, we test the performance of S-PDLA with a real application, called ''Anonymous LANL App 2'' [23]. We choose this application because it has complex access patterns, which can evaluate the generality of our proposed approach.

In this application, each process issues I/O requests with various access patterns at different parts of a shared file during the run time. We can divide the file into three typical parts according to the request sizes. For the first part, the request sizes are relatively small (several KBs). The second part has medium request sizes (tens of KBs). For the last part, the request sizes change between a large one (hundreds of KBs) and another large one (hundreds of KBs) iteratively. To simulate the same access scenario, we replay the data accesses of the application according to the I/O trace.

Figure 13a shows the corresponding I/O performance of the application under the four replication policies. Similar to the previous tests, while S-ORIG, S-PDLA and NS-PDLA outperform H-PDLA individually, S-PDLA brings the best performance for the highly varied workload of the application. The reason is that S-PDLA replicates different patterns on the proper servers with optimized data layouts based on performance benefit analysis, while S-ORIG places the data on SServers with the default data layout and NS-PDLA always indiscriminately replicates them on SServers. This result shows that S-PDLA is also efficient for applications with complex I/O patterns.

5.3 System Overhead

As shown in Figure 1, we integrate some components into the default parallel I/O system to make PDLA work automatically. These modules cause additional system overheads.

5.3.1 Trace Collection Overhead

We collect I/O traces in the first execution of the application. It only happens once. To measure the run-time overhead of the trace collector, we test the difference between the execution times of IOR and MPI-Tile-IO with and without linking the tracing library. For IOR benchmark the overhead is below 2%, and for MPI-TILE-IO benchmark it is below 6%. These results show that the overhead introduced by the trace collector is acceptable.

5.3.2 Pattern Identification Overhead

We conduct access pattern recognition offline in the background. Thus it does not affect the execution of user applications. The execution time of the analyzer is proportional to the number of traced records. That is, its time complexity is O(n). To show the overhead, we run the analyzer with a PIO-Bench trace file with 5296 records on a single core in a client node. The result shows it takes less than 2 seconds to finish the analyzing. We can parallelize the analysis process to reduce the execution time if necessary. For example, we can use shell scripts to start multiple analyzers on multiple trace files simultaneously.

5.3.3 I/O Redirection Overhead

For some applications with recognized data access patterns and replica files, the system can improve the overall I/O performance. However, some applications do not have regular data access patterns thus will not benefit from the accesses to the PDLA replica files. In this case, overhead may exist, which may degrade performance if it is noticeable in volume.

In this subsection, we only evaluate the following two possible sources of overheads in the runtime system.

1) During "file open", the I/O redirection module needs to look up the data access pattern in the Pattern Database.

2) During "file read/write", the module needs to check whether the opened file is a replica, thus to decide whether to do the offset calculation.

To evaluate the overhead in the runtime, we run IOR with contiguous data access pattern, and put no related pattern in the database and make no data replication. So the system just accesses the original files. We run IOR with 8, 64, and 512 processes, and each of them 100MB data with the request size of 256KB. Figure 13b shows the execution time's percentage change after enabling H-PDLA comparing with the original I/O system. We can see that the overhead of H-PDLA is acceptable. For S-PDLA, it also incurs performance overhead due to the same reasons for H-PDLA. Since this overhead is comparable to that of H-PDLA, we omit it in Figure 13b for concise illustration.

6 Related Work

Data replication: There are many efforts to utilize data replication for improved I/O system performance. InterferenceRemoval [24] replicates segments of files that could be involved in interference to their respectively designated I/O nodes, so that the degree of interference on each node can be greatly reduced. IR+ [25] uses SSDs to create disk-friendly layouts for data replicas in heterogeneous storage

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

systems. Different from these works, PDLA creates replicas based on data access patterns thus the rule of selecting data to be replicated is straightforward. PDLA also reorganizes data according to the access order, so it transforms non-contiguous accesses to contiguous ones.

To enhance the I/O efficiency of non-contiguous requests, HDFS [18] and GPFS-SNC [26] create multiple copies for each file, so that I/O requests can be redirected to the nearest location to reduce data access costs. Similar approaches are adopted in the cluster or grid systems [27], [28]. In contrast to these works, PDLA only replicates access patterns thus it can save storage space. Pattern-based data replication is also conducted in [29], [30], which select general access patterns and replicates them with more reorganized layouts. CEDA [31] considers the physical data layout of file requests and carries out cost-effective data replication. However, all these studies utilize the same type of devices in the original I/O system to replicate data, which may still have performance limitation for applications with high I/O concurrency. As opposed to them, S-PDLA leverages new storage media, SSDs, to improve I/O performance.

Besides, some parallel file systems, such as Ceph, Lustre, and GPFS, provide built-in data replication functionalities for enhanced fault tolerance. Similar redundant data placement approaches [32], [33] are designed to improve system availability. In contrast to these efforts, this study intends to boost system performance via data replication, an area that has attracted attention only recently [8].

Data organization: AILS [34], FS2 [35], and BORG [36] automatically reorganize selected disk blocks to reduce disk seeking overhead. These techniques are efficient for single disk and disk arrays but require complex implementation in the disk device driver and local file systems. With a simple implementation in I/O middleware, PDLA replication scheme suits today's large-scale HPC systems well and has better pattern recognition ability.

To improve read performance, SOGP [37] stores a copy of data that is often accessed in a more efficient organization. While SOGP aims to bridge the gap between OrangeFS and local storage, PDLA focuses on bridging the gap between application and physical data. He et al. proposed a file reorganization method according to access pattern to increase the continuity by remapping files in MPI-IO layer [38], [39]. Compared with that, PDLA consumes less storage resource and is more flexible for further data layout optimization. More importantly, PDLA is a combined system approach with replication, reorganization, and optimized data layout working collectively for best performance.

For write optimization, PLFS [40] stores the data of non-contiguous writes in a set of reorganized log files. While dramatically improving write performance, it may degrade read performance when reading back from those files due to the inevitable data restructuring. In the PDLA scheme, read and write access patterns are handled separately to achieve optimized performance for both of them.

7 Conclusion

PDLA is a new data replication scheme aiming to optimize parallel I/O performance based on application-specific I/O characteristics. It includes an HDD-based scheme *H-PDLA* and an SSD-based scheme *S-PDLA*. For applications with relatively low I/O concurrency, H-PDLA identifies the access patterns of applications and makes a reorganized data replica for each access pattern on HDD-based servers with an optimized data layout. For applications with high I/O concurrency, S-PDLA identifies the critical access patterns and replicates them on SSD-based servers or all HDD-based and SSD-based servers with optimized layouts. Experimental results show that (1) H-PDLA can significantly improve the performance of the original I/O system and (2) S-PDLA can further improve the system performance of H-PDLA for applications with a large number of processes.

In the future work, we plan to utilize more application access patterns to make storage systems more intelligent and more efficient. We also intend to test our proposed replication scheme with more applications from both performance and energy consumption point-of-view.

Acknowledgment

This work was supported in part by the National Science Foundation of China No. 61572377, the Fundamental Research Funds for the Central Universities No. 2018QNA5015, and the Zhejiang Lab Research Project No. 2019KC0AC01.

References

[1] S. He, X.-H. Sun, and B. Feng, "S4D-Cache: Smart Selective SSD Cache for Parallel I/O Systems," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2014, pp. 514 – 523.

[2] D. Chen, Y. Hu, L. Wang, A. Y. Zomaya, and X. Li, "H-PARAFAC: Hierarchical Parallel Factor Analysis of Multidimensional Big Data," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 4, pp. 1091–1104, 2017.

[3] Cebit.com.au, "Square Kilometre Array (SKA) Telescope Will Generate Big Data," http://tinyurl.com/8wlzl9o, 9 August 2012.

[4] S. He, X.-H. Sun, and A. Haider, "HAS: Heterogeneity-Aware Selective Data Layout Scheme for Parallel File Systems on Hybrid Servers," in *Proceedings of 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015, pp. 613–622.

[5] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[6] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O Accesses through MPI-IO," in *Proceedings of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2003.

[7] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File Systems," in *Proceedings of IEEE International Conference on Cluster Computing*, 2003.

[8] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A Cost-Intelligent Application-Specific Data Layout Scheme for Parallel File Systems," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC)*, 2011, pp. 37–48.

[9] S. He, Y. Wang, and X.-H. Sun, "Boosting Parallel File System Performance via Heterogeneity-Aware Selective Data Layout," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. PP, no. 99, pp. 1–1, 2015.

[10] T. Madhyastha and D. Reed, "Exploiting Global Input/Output Access Pattern Classification," in *Proceedings of IEEE/ACM Supercomputing '97*, 1997.

[11] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting Application-Specific Parallel I/O Optimization Using IOSIG," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2012, pp. 196–203.

[12] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008, pp. 1–12.

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2946135, IEEE Transactions on Computers

[13] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011, pp. 22–32.

[14] S. He, X.-H. Sun, and Y. Wang, "Improving Performance of Parallel I/O Systems through Selective and Layout-Aware SSD Cache," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. PP, no. 99, p. 1, 2016.

[15] S. He, X.-H. Sun, Y. Wang, and C. Xu, "A Migratory Heterogeneity-Aware Data Layout Scheme for Parallel File Systems," in *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, Conference Proceedings, pp. 1133–1142.

[16] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, "Pattern-Direct and Layout-Aware Replication Scheme for Parallel I/O Systems," in *Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013, pp. 345–356.

[17] S. He, X.-H. Sun, Y. Wang, A. Kougkas, and A. Haider, "A Heterogeneity-Aware Region-Level Data Layout Scheme for Hybrid Parallel File Systems," in *Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, 2015.

[18] D. Borthakur, "HDFS Architecture Guide," *Hadoop Apache Project*, 2008.

[19] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1999, pp. 183–191.

[20] "Interleaved Or Random (IOR) Benchmarks," 2017. [Online]. Available: http://sourceforge.net/projects/ior-sio/

[21] "PIO-Bench," 2017. [Online]. Available: ftp://ftp.parl.clemson.edu/pub/pio-bench/

[22] "MPI-Tile-IO Benchmark," 2017. [Online]. Available: http://www.mcs.anl.gov/research/projects/pio-benchmark/

[23] "Application I/O Traces: Anonymous LANL App2," http://institutes.lanl.gov/plfs/maps/, 2014.

[24] X. Zhang and S. Jiang, "InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication," in *Proceedings of IEEE/ACM Supercomputing*, 2010.

[25] X. Zhang, S. Jiang, A. Diallo, and L. Wang, "IR+: Removing Parallel I/O Interference of MPI Programs via Data Replication over Heterogeneous Storage Devices," *Parallel Computing (ParCo)*, 2018.

[26] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, and D. Subhraveti, "GPFS-SNC: An Enterprise Storage Framework for Virtual-Machine Clouds," *IBM Journal of Research and Development*, 2011.

[27] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, and I. Foster, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," in *Proceedings of Mass Storage Conference*, 2001.

[28] A. Chakrabarti and S. Sengupta, "Scalable and Distributed Mechanisms for Integrated Scheduling and Replication in Data Grids," *Distributed Computing and Networking*, pp. 227–238, 2008.

[29] J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. Ross, and N. F. Samatova, "RADAR: Runtime Asymmetric Data-Access Driven Scientific Data Replication," in *Proceedings of the International Supercomputing Conference (ISC)*. Springer, 2014, pp. 296–313.

[30] H. Tang, S. Byna, S. Harenberg, X. Zou, W. Zhang, K. Wu, B. Dong, O. Rubel, K. Bouchard, S. Klasky, and N. F. Samatova, "Usage Pattern-Driven Dynamic Data Layout Reorganization," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 16-19 May 2016 2016, pp. 356–365.

[31] S. He and X.-H. Sun, "A Cost-Effective Distribution-Aware Data Replication Scheme for Parallel I/O Systems," *IEEE Transactions on Computers (TC)*, vol. PP, no. 99, pp. 1–1, 2018.

[32] A. Brinkmann, S. Effert, C. Scheideler *et al.*, "Dynamic and Redundant Data Placement," in *Proceedings of International Conference on Distributed Computing Systems*, 2007.

[33] A. Brinkmann and S. Effert, "Redundant Data Placement Strategies for Cluster Storage Environments," *Principles of Distributed Systems*, pp. 551–554, 2008.

[34] W. Hsu, A. Smith, and H. Young, "The automatic improvement of locality in storage systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 4, pp. 424–473, 2005.

[35] H. Huang, W. Hung, and K. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 263–276, 2005.

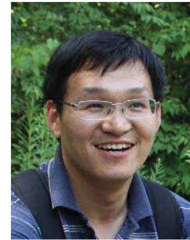
[36] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-Optimizing Storage Systems," in *Proccedings of the 7th USENIX Conference on File and Storage Technologies*, 2009.

[37] P. Gu, J. Wang, and R. Ross, "Bridging the Gap between Parallel File Systems and Local File Systems: A Case Study with PVFS," in *Processings of International Conference on Parallel Processing*, 2008.

[38] J. He, H. Song, X.-H. Sun, Y. Yin, and R. Thakur, "Pattern-Aware File Reorganization in MPI-IO," in *Proceedings of Parallel Data Storage Workshop*, 2011.

[39] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT press, 1999, vol. 1.

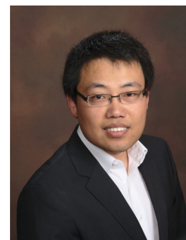
[40] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A Checkpoint Filesystem for Parallel Applications," in *Proceedings of IEEE/ACM Supercomputing (SC)*, 2009.



Shuibing He received the PhD degree in computer science and technology from Huazhong University of Science and Technology, in 2009. He is now a ZJU100 Young Professor in the College of Computer Science and Technology at Zhejiang University. His research areas include parallel I/O systems, file and storage systems, high-performance and distributed computing. He is a member of the IEEE and ACM.

Yanlong Yin received his PhD degree in Computer Science from Illinois Institute of Technology in 2014. Before that, he received his BE degree in Computer Science and MS degree in Computer Architecture from Huazhong University of Science and Technology, Wuhan, China in 2006 and 2008. Yanlong is currently a research scientist in Zhejiang Lab. His research interests include parallel computing systems, parallel I/O systems, and parallel file systems.

Xian-He Sun received his PhD degree in Computer Science in 1990 from Michigan State University. He is currently a distinguished professor of the Department of Computer Science, the Illinois Institute of Technology (IIT). His research interests include parallel and distributed processing, memory and I/O systems, and performance evaluation. He is an IEEE fellow.



Xuechen Zhang received the MS and the PhD in Computer Engineering from Wayne State University. He is currently an assistant professor in the School of Engineering and Computer Science at Washington State University Vancouver. His research interests include the areas of file and storage systems and high-performance computing. He is a member of the IEEE and ACM.

Zongpeng Li received his BE in Computer Science from Tsinghua University in 1999, and his PhD from University of Toronto in 2005. Since 2005, he has been affiliated with University of Calgary and then Wuhan University. His research interests include computer networks, cloud computing and computer systems.

0018-9340 (c) 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.