A Cost-Effective Distribution-Aware Data Replication Scheme for Parallel I/O Systems

Shuibing He and Xian-He Sun, *Fellow, IEEE*

Abstract—As data volumes of high-performance computing applications continuously increase, low I/O performance becomes a fatal bottleneck of these data-intensive applications. Data replication is a promising approach to improve parallel I/O performance. However, most existing strategies are designed based on the assumption that contiguous requests are being served more efficiently than non-contiguous requests, which is not necessarily true in a parallel I/O system. The reason is that the multiple-server data distribution makes the favorable accesses between contiguous requests and non-contiguous ones indeterminate. In this study, we propose CEDA, a cost-effective distribution-aware data replication scheme to better support parallel I/O systems. As logical file access information is inefficient to make replication decisions in a parallel environment, CEDA considers physical data accesses on servers in both data selection and data placement during a parallel replication process. Specifically, CEDA first proposes a distribution-aware cost model to evaluate the file request time with a given data layout, and then it carries out cost-effective data replication based on replication benefit analysis. We have implemented CEDA as a part of the MPI I/O library in light of high portability on top of the OrangeFS file system. By replaying representative benchmarks and a real application, we collected comprehensive experimental results on both HDD- and SSD-based servers and conclude that CEDA can significantly improve parallel I/O system performance.

Index Terms—Parallel I/O system, parallel file system, data layout, data replication, data reorganization

1 Introduction

In the big data era, high-performance computing (HPC) applications in scientific and engineering fields are becoming extremely data-intensive. For example, many of the INCITE applications at Argonne Leadership Computing Facility (ALCF) generate more than one petabyte (PB) of data in a single run [1]. The data generated by the U.S. Department of Energy (DOE) leadership-computing facilities is projected to exceed one exabyte (EB) per year by 2018 [2]. With the ever increasing data requirements, I/O performance has become the fatal bottleneck of many data-intensive HPC applications [3], [4], [5].

To overcome this I/O bottleneck issue, modern HPC clusters deploy parallel I/O systems to provide data storage services. As shown in Fig. 1, parallel I/O system often includes several layers, such as application (App), I/O middleware, parallel file system (PFS), and storage hardware layer. The PFS, such as OrangeFS [6], Lustre [7], and GPFS [8], usually distributes user data on multiple servers, physically connecting each via network. By exploiting the parallelism of multiple storage devices, a parallel I/O system can provide largely increased storage capacity and I/O bandwidth.

- S. He is with the School of Computer Science, Wuhan University, Wuhan, Hubei 430072, China. E-mail: heshuibing@whu.edu.cn.
- X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois 60616. E-mail: sun@iit.edu.

Manuscript received 29 July 2017; revised 22 Feb. 2018; accepted 24 Apr. 2018. Date of publication 29 Apr. 2018; date of current version 14 Sept. 2018.
(Corresponding author: Shuibing He.)
Recommended for acceptance by G. Min.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2831689

Although parallel I/O systems promise decent peak performance for large requests, they do not always deliver desirable performance even for some commonly-used access patterns. For example, when facing non-contiguous small requests, a PFS usually performs poorly [3], [9], [10], [11]. Even with contiguous large requests, the performance of a PFS can be seriously downgraded due to the I/O interference when a large number of processes concurrently access data [12].

Data replication has emerged as a promising method to improve I/O system performance. By replicating user data in the unused storage space (replica space) with reorganized layouts, original I/O requests can be redirected to new locations with optimized access patterns. There are two critical issues for data replication: (1) deciding which data need to be replicated, and (2) how to place them efficiently in the replica space. As sequential access on disk usually leads to higher I/O efficiency, most existing strategies select non-contiguously accessed data and contiguously place them in the replica space to improve I/O system performance [13], [14], [15], [16], [17]. For example, PDLA [15] selects non-contiguously accessed data on a file and distributes them contiguously in the replica files. RADAR [16] and Dynamic Reorganization [17] create one or multiple replicas for sub-region accesses and element selections.

Despite various advantages of these data replication schemes, all the strategies are designed based on the assumption that contiguous requests are being served more efficiently than non-contiguous ones. However, this assumption is usually true in a serial I/O system with a single disk. For a parallel I/O system consisting of multiple disks, this assumption does not always hold true due to the following reason.

In a parallel I/O system, a logical file is usually distributed on multiple servers with a specific data layout, in terms of

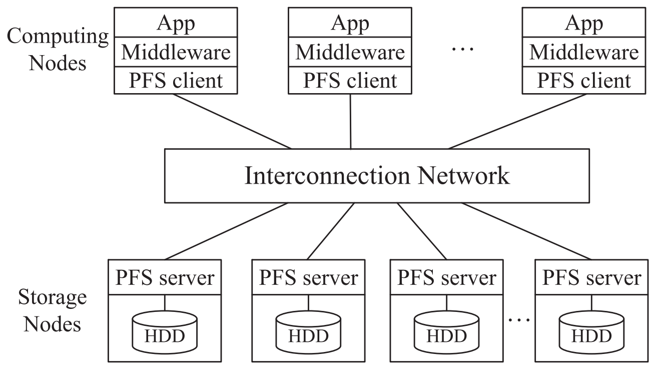


Fig. 1. Architecture of a typical parallel I/O system.

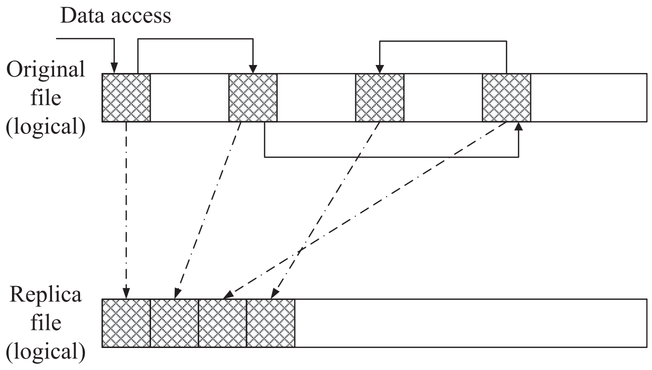


Fig. 2. Traditional data replication scheme, which usually selects the non-contiguously accessed file requests and concatenates them together in the replica space.

stripe size, stripe factor, and number of servers. This multiple-server distribution makes the favorable accesses between contiguous and non-contiguous file requests uncertain. On the one hand, as non-contiguous file requests may be mapped to a single sever and be accessed sequentially on disks, the I/O performance of non-contiguous file requests can be largely enhanced. On the other hand, contiguous file requests may be divided into multiple non-contiguous sub-requests on servers, which will largely decrease the efficiencies of disks as the physical data access continuity is not maintained. As a consequence, traditional data replication schemes that based on the above assumption will experience considerable performance inefficiencies in a parallel I/O system.

In this paper, we propose CEDA, a cost-effective distribution-aware data replication scheme to boost parallel I/O system performance. As logical file access information remains inefficient to make replication decisions in a parallel I/O system, CEDA considers physical data accesses on servers in both data selection and data placement decisions during a parallel replication process. More specifically, to make the approach truly effective, CEDA first introduces a data cost model that considers physical data distribution to evaluate the access time of a file request with a given data layout. Then with this model, CEDA makes data replication decisions based on data access cost analysis: as the size of replica space is limited, CEDA only carries out the cost-effective data replication that can obtain performance benefits to maximize the system I/O performance. Furthermore, CEDA resides in the I/O middleware layer and is transparent to applications and file system(s), thereby it requires no or minimal modification to parallel I/O stack implementations and can be applied to several parallel file systems.

Specifically, we make the following contributions.

- We find that contiguous file requests do not necessarily lead to optimized I/O performance over non-contiguous file requests in a parallel I/O system. Instead, physical data access continuity on server and inter-server load balance do.
- We introduce a distribution-aware cost model for parallel I/O systems, which can evaluate the I/O access times of file requests by considering physical data accesses on servers with a given file system data layout.
- We propose a cost-effective data replication scheme, which first evaluates whether a potential data replication is beneficial based on the proposed cost model, and then only carries out the cost-effective data replication that can actually boost I/O system performance.
- We implement CEDA under MPICH2 I/O library [18] on top of OrangeFS PFS. Our extensive experiments on both HDD- and SSD-based I/O systems with well-known benchmarks, such as IOR, HPIO and BTIO, and a real application confirm the performance and scalability benefits of CEDA.

The remainder of this paper is organized as follows. Section 2 discusses the background and motivation. Section 3 introduces the data access cost model. The design and implementation of CEDA are described in Sections 4 and 5. Section 6 presents the performance evaluation. Section 7 discusses the related work. Finally, we conclude this paper in Section 8.

2 Background and Motivation

2.1 Logical Continuity-Based Data Replication

To address the I/O bottleneck issue, data replication has emerged as a promising method to speed up I/O accesses. Fig. 2 illustrates the idea of traditional data replication scheme [15], [16], [19], which usually chooses the non-contiguously accessed data from the original file system and places them continuously in the replica space, so that original non-contiguous file requests can be transformed to contiguous file requests. As we have mentioned previously, although this approach can help to improve I/O performance in certain cases, it may suffer from considerable inefficiencies in a parallel I/O system, as we will illustrate in the following Section 2.2.

2.2 A Motivating Example

To demonstrate this problem, we run IOR [20] with sequential I/O requests and strided I/O requests on an OrangeFS file system on four servers. For both access patterns, IOR runs with four processes, and the request size ranges from 4 to 64 KB. For the sequential I/O pattern, each process issues sequential I/O requests to access the $1/n$ space of a shared file, where n is the number of the processes. For the strided I/O pattern, the stride size is $(n-1)$ times of the request size.

Fig. 3 shows the I/O bandwidth of the system with sequential I/O patterns and strided I/O patterns at various request sizes. The bandwidth is measured as the aggregated data amount of IOR divided by the I/O completion time. As

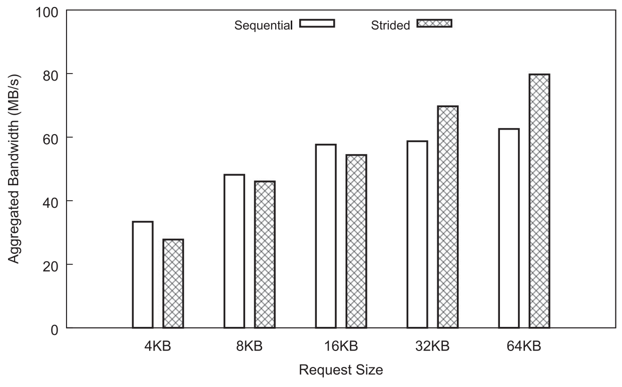
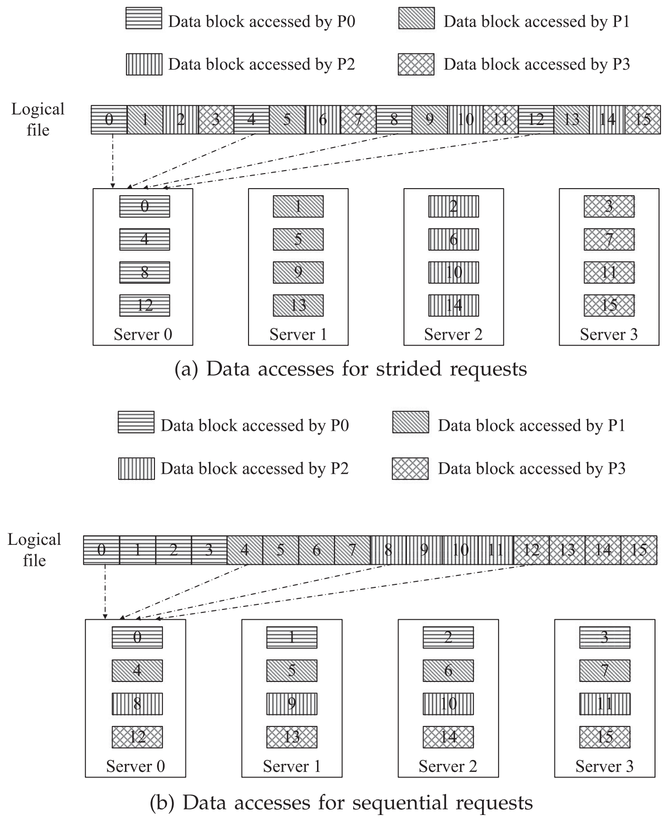


Fig. 3. Bandwidths of IOR with strided requests and sequential requests. This figure shows non-contiguous requests may be more efficient than contiguous ones.



Data block accessed by P0
Data block accessed by P1
Data block accessed by P2
Data block accessed by P3
Logical file
Server 0
Server 1
Server 2
Server 3
(a) Data accesses for strided requests
(b) Data accesses for sequential requests

Fig. 4. Illustration of how data are accessed on multiple servers in a parallel I/O system.

expected, with sequential requests, increasing the request size leads to increased bandwidth. This is also true for strided I/O requests. However, although sequential requests have better bandwidths over strided requests at some request sizes, e.g., 4, 8, and 16 KB, we find that the bandwidth for strided requests is much higher than that for sequential requests at some other request sizes, e.g., 32 and 64 KB. This appears to be inconsistent with the assumption that contiguous I/O requests will be more efficiently serviced than non-contiguous I/O requests.

The reason for this phenomenon lies in the fact that a file will be distributed on multiple servers, so that the order in which the requests arrive at servers is different from the original file request order. Fig. 4 illustrates the different orders when strided I/O requests (Fig. 4a) and sequential I/O requests (Fig. 4b) are issued to the PFS at request size 64 KB. When strided I/O pattern is used, each process sends four non-contiguous requests in the order of their offsets in the logical file space. Because both request size and stripe size are 64 KB, the four non-contiguous logical requests issued by a process are sent to a particular server. Because the local file system generally allocates data on the disk in an order that is consistent with their offsets in logical file space, the consequent sequential service order at a server leads to an effective sequential data access at the server, which leads to high I/O performance. On the contrary, when sequential I/O pattern is used, the four requests at each server are from four concurrent processes and arrive in an order determined by the relative progress of the processes, which is unpredictable [12]. Therefore, a server usually serves requests in a random order, substantially degrading the server performance.

These observations indicate that contiguous file requests are not always more efficient than non-contiguous ones in parallel I/O systems, thus blindly choosing non-contiguous file requests and replicating them contiguously in the replica space may experience considerable inefficiencies. As can be seen from Fig. 4, the physical data access continuity on each disk and inter-server load balance can actually affect the I/O system performance. This motivates us to propose our cost-effective distribution-aware data replication scheme to better support parallel I/O systems.

2.3 Pattern-Based Data Replication

Several previous studies replicate all file system data with one or multiple copies [21], [22]. Although the space cost is not a big deal for a small file system, it may be a severe concern for a large-scale parallel file system due to the ever increasing data sizes. For example, the data amount of the *Quantum Chromodynamics* application in ALCF [1] exceeds 1 PB. This means that 200 10TB additional disks are needed if we adopt a 3-way replication policy. The acquisition cost and power management overhead will put high pressure for most HPC systems.

An alternate approach is to avoid the full file system data replication. Fortunately, many data-intensive applications in HPC domains have predictable I/O access patterns [15], [23], [24], [25], [26], which enables partial data replication. This is because HPC applications often run multiple times, and their data access patterns are mostly determined by their inherited numerical methods, not input data. For example, the BTIO application [27] that solves block-tridiagonal matrices has this feature. Once the parameters, such as size of the array, number of time steps, etc., are given, the I/O behaviors of BTIO can be accurately predicted. Due to the iterative loops of program codes, many applications have repeated data access patterns in one execution [28]. Furthermore, the checkpointing technique that is widely-used in HPC machines to improve system reliability also has repetitive access patterns [29].

Based on these observations, it is reasonable for CEDA to perform pattern-based partial data replication rather than replicating all the data in file system. Similar approaches, which utilize I/O access patterns to optimize I/O system performance, have been successfully used many times in data partition [30], data replication [15], and data prefetching [31].

TABLE 1
Parameters in the Cost Analysis Model

System Parameters	
m	Number of servers in the system
α	Average startup time of one I/O operation
β	Data transfer time of one unit of data
Application Pattern Parameters	
p	Number of client processes
o	Offset of file request
l	Length of file request
Data Layout Parameters	
str	Stripe size on each server
g	Number of groups to divide the servers
f	Stripe factor
n	Number of servers used to store file ($n \leq m$)

3 Distribution-Aware Data Access Time Cost Model

As logical file access information may mislead replication decisions in parallel I/O systems, we propose a cost model to evaluate the data access time of a file request by considering physical data distribution on servers. The corresponding parameters are listed in Table 1. This model is inspired by the previous work [21], but with two differences. First, the file request does not need to be uniformly distributed on multiple servers, so that it can accommodate more general access patterns. Second, it considers data accesses on each single server and load balance among multiple servers, so that it can efficiently evaluate I/O system performance.

As a parallel file request may involve multiple sub-requests, we first describe the data access cost of a single sub-request, then derive the access time of a single server by considering the sub-requests distributed on them, and finally calculate the data access cost of all file requests.

Sub-Request Access Cost. For each sub-request served by a file server, the access cost is defined as the I/O completion time of the sub-request, which can be represented as

$$T_{sub} = T_s + T_t, \tag{1}$$

where T_s denotes the storage startup time before an actual I/O operation, and T_t stands for the storage transfer time spent on actual data read/write operation on the device [3].

T_s is mainly determined by the number of seeks and software overhead on the device. If the data access is a contiguous operation, then the startup time can be omitted, i.e., $T_s = 0$. Otherwise, the access requires one disk seek operation, thus $T_s = \alpha$. T_t is proportional to the data size of the sub-request on the server. Given the logical file access and data layout information in Table 1, we can obtain the sub-request size on each server as follows.

Assuming that the file is distributed on server 0 to $n-1$ in a round-robin fashion, the offset and the length of the ith file request are o and length l, then the serial number of the involved beginning and ending server are $J = \lfloor \frac{o}{str} \rfloor \% n$ and $K = \lfloor \frac{o+l}{str} \rfloor \% n$. The file request may also incur fragments, and the size of the beginning and ending stripe fragment are $b = str - l\%str$ and $e = (o+l)\%str$. According to the fragment distribution on server j, Fig. 5 shows the five possible

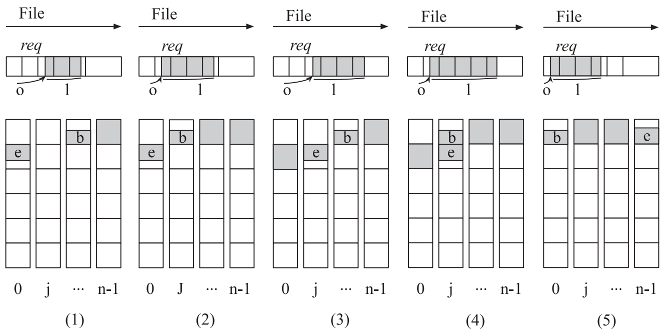


Fig. 5. Five possible sub-request distribution cases on servers. (1): There is no sub-request on server j; (2): Only the beginning fragment is on server j; (3): Only the ending fragment is on server j; (4): Both the beginning and ending fragment are on server j; (5) There is no fragment on server j.

sub-request distribution cases on servers. Let $\triangle = \lfloor \frac{o+l}{str} \rfloor - \lfloor \frac{o}{str} \rfloor$, then the sub-request size of the ith file request on server j can be calculated as follows:

$$s_{ij} = \begin{cases} 0, & j < J \text{ and } \triangle < n \\ b + \triangle/n * str, & j = J \text{ and } j \neq K \\ e + \triangle/n * str, & j \neq J \text{ and } j = K \\ b + e + \triangle/n * str, & j = J = K \\ \lceil \triangle/n \rceil * str, & otherwise. \end{cases} \tag{2}$$

The s_{ij} is calculated with the simple round-robin layout. For other given data layouts, the calculation is similar but with a different value. We omit it here due to space limitations.

Based on the value of s_{ij}, the storage transfer time of the sub-request can be calculated as $T_t = s_{ij} * \beta$.

Server Access Cost. The data access time on server relies on the overall access time of all sub-requests on the server.

The startup time is determined by the overall startup time of all sub-requests. If there is only one process, then the number of seeks equals the number of non-contiguous sub-requests from the process. Assume that the number is denoted by k, then $T_s = k\alpha$. Otherwise, it is a random variable as all sub-requests will arrive at the server in a random order. Assuming that p is the number of processes and q is the number of sub-requests on the server, then in the worst case each sub-request involves one seek operation and in the best case each process needs one seek. It is reasonable to assume that the number of seeks has a uniform distribution between the two cases. That is, $T_s = \frac{1}{2}(p+q)\alpha$.

The data transfer time on server is determined by the overall data size on that server. Assuming that the system has N file requests, then the overall data transfer time on the jth server is $\sum_{i=1}^{N} s_{ij} * \beta$.

Putting it all together, the access cost of server j is

$$T_{ser}^{j} = \begin{cases} k\alpha + \sum_{i=1}^{N} s_{ij} * \beta, & p = 1 \\ \frac{1}{2}(p+q)\alpha + \sum_{i=1}^{N} s_{ij} * \beta, & otherwise. \end{cases} \tag{3}$$

File System Access Cost. As all file requests are concurrently served by multiple servers, the data access time of the file system is determined by the slowest server. Assuming that the access cost of each server is T_{ser}^{j}, where $0 \leqslant j \leqslant (n-1)$, then the data access cost of the file system is described as follows:

$$T_{sys} = \max\{T_{ser}^{0}, T_{ser}^{1}, \ldots, T_{ser}^{n-1}\}. \tag{4}$$

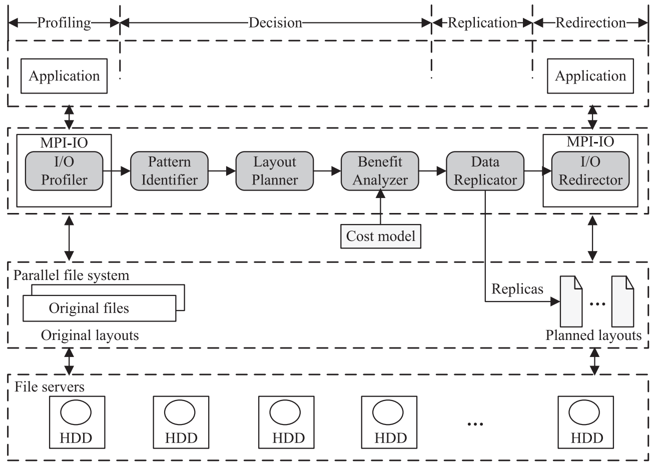


Fig. 6. System architecture of the cost-effective distribution-aware data replication scheme.

Note that the cost model applies to both HDD- and SSD-based file servers. In both cases, the model is similar except the startup time and the data transfer time for HDD and SSD will change, as we discussed in our previous work [25], [26]. Generally, HDD has much larger startup time and data transfer time compared to SSD, due to their different storage characteristics.

Discussion and Guideline. From the analysis of the cost model, we can see that the file system performance depends on the data access patterns and the data distribution manners. Generally, data access patterns and data distribution manners have an interrelated effect on I/O system performance. Hence, we can obtain some brief guidelines for data replication as an effective approach to reorganize data layout.

- If the number of non-contiguous sub-requests on a server, namely the value of k is reduced, the overall cost can be benefited.
- Decreasing the number of processes involved in a single server would also be helpful for the overall system performance.
- Distributing the accesses evenly on multiple servers could mitigate the risk of an over-loaded server limiting system performance.

4 Design of CEDA

With the proposed model, we can calculate the I/O time of file requests with a given data layout. If we have some prior knowledge about application's file requests, then we can evaluate whether the new data layout created by the data replication scheme can improve the original I/O system performance. Fortunately many applications have predictable data access patterns [15], [23], as previously discussed in Section 2.3. Based on this fact, CEDA first identify the I/O access patterns of an application in its first run, and then it performs cost-effective data replication based on the model to speed up the application in its subsequent runs.

4.1 Overview of CEDA

Fig. 6 shows the system architecture of CEDA. In these systems, application processes on compute nodes access the data on file servers by calling the middleware library (MPI-IO). CEDA resides in the MPI-IO library; it is not only responsible for selecting data from the original file system, but also replicating them in the replica space that is within the same file system. By applying cost-effective distribution-aware data replication, CEDA can improve parallel I/O system performance.

Data replication can be built at different levels in the parallel I/O stack, we position CEDA in the middleware layer for the following reasons. First, key global access information, such as file-level, process-level, and MPI Rank-level attributes are accessible. This enables I/O optimization with global access information. Second, the middleware layer is independent of the underlying layer, allowing the solution to support multiple parallel file systems, such as PVFS [32], OrangeFS [6], and Lustre [7]. Third, the design is transparent to applications, therefore no or minimal modification is required for user programs to utilize the increased performance.

The whole procedure of CEDA consists of four phases. In the "profiling phase", the *profiler* collects the I/O access information of the application and stores them into traces during the application's first run. In the "decision phase", the *identifier* takes the traces as inputs and exacts access patterns of the application. Then, the *planner* creates a planning data layout to store the replica data of each identified access pattern. Next, with the original and the planning data layout, the *analyzer* evaluates the performance benefit of each data replication based on the proposed cost model. Once the replication is cost-effective, in the "replication phase" the *replicator* carries out the actual data movement for each access pattern, by copying selected data from the original files to their replicas. Finally, in the "redirection phase" the *redirector* directs I/O requests either to the original files or the replicas based on the specifics of the requests during the subsequent runs of the application.

4.2 I/O Pattern Identifying

I/O patterns refer to the access behaviors of applications in multiple dimensions. The spatial pattern defines the access distance between successive requests, and the temporal pattern represents the time intervals between two requests. There are also other patterns can be defined in our scheme. The most important pattern is the spatial pattern, as it has a great potential to improve I/O system performance. Hence, in this study we focus on but are not limited to the spatial patterns.

To obtain user's access patterns, the *profiler* captures the I/O activities of the application in the first run. Although a number of tools can be used, we use IOSIG [31] to profile applications because it can serve our purpose and only introduce acceptable overhead [33]. IOSIG stores the I/O access information, such as process ID, MPI rank, file descriptor, type of operation, offset, request size, and time stamp information in several traces.

The *identifier* takes the traces as input to generate the access patterns. It first separates the trace entries by MPI rank, each for one process. Then, it scans the I/O traces chronologically and attempts to derive the access patterns by utilizing a template matching approach. More details about the pattern identification can be found in the prior work [31]. Finally, once the analyzing process finishes, the obtained access patterns are stored in a global pattern database for further use.

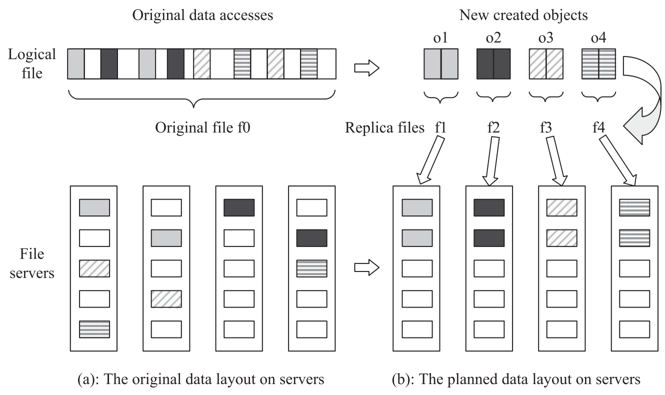


Fig. 7. An example of the data layout before and after replication. In the original layout, each server has non-continuous accesses and the I/O load of each server is three, two, one, and two blocks, respectively; in the planning layout, each server has two contiguous accesses, each with one block. The planning layout provides enhanced physical data access continuity on disks and inter-server load balance.

4.3 Pattern-Based Replica Layout Plan

The *planner* is responsible for creating data layout plan for identified data access patterns before they are actually replicated. As logical file access does not ultimately determine the parallel I/O performance, the goal of the *planner* includes three enhancements: (1) physical access continuity on each disk, (2) I/O parallelism on servers, (3) inter-server load balance.

Algorithm 1 shows the detailed procedure to generate the planning data layout for each access pattern. It first concatenates the requested data of each process into a contiguous logical object, no matter whether the access pattern is contiguous or not. Next, it creates n replica files, each on a separated server, to store the objects from all processes, where n is the number of the servers. Finally, it uses a modulo operation to determine file l where an object i resides (l = i mod n). If multiple objects are mapped into the same file, then the objects are distributed into the file in an ascending order of MPI Rank where the object belongs. With these efforts, the planning data layout tries to achieve the goal of the *planer*.

Algorithm 1. The Planning Layout Generating Algorithm

```
1:  procedure LAYOUT_PLANNER($Seq, RGT, RMT$)
2:      for each sequence in $Seq$ do                ▷ Create objects
3:          Create an object $o_i$ for the $i$th process
4:          for each request $r_j$ in $Seq[i]$ do
5:              if $r_j$ is $\notin$ $o_i$ then
6:                  Append $req_i$ to the end of $o_i$
7:              end if
8:          end for
9:      end for
10:     for each server $S_k$ in $S[1, n]$ do         ▷ Create replicas
11:         Create a file $f_k$ on $S_k$
12:     end for
13:     for each object $o_i$ do                 ▷ Place objects on replicas
14:         $l \leftarrow i \bmod n$
15:         Append object $o_i$ to the end of $f_l$
16:     end for
17: end procedure
```

Fig. 7 is an illustrative example to show how the *planner* generates the planning data layout for one access pattern. In this example, there are four processes, each sending two non-contiguous requests to a shared file $f0$. The *planner* gathers requests from the same process and sequentially rearranges them into an object for each process. Then the four objects $o1$ to $o4$ are placed into four replica files $f1$ to $f4$, each residing on one server. Compared to the case before replication, the new data layout makes the disk on each server serve less process' requests in a more continuous order. At the same time, the I/O load on each server is largely balanced, which also benefits the I/O performance.

4.4 Data Replication Benefit Analysis

With the planning data layout for replicas, the *analyzer* can determine whether an actual data replication can improve I/O system performance. This is conducted by executing replication benefit analysis based on the proposed cost model. It considers two data layouts for each access pattern. The first one is the original data layout, which distributes the file data over multiple servers in the original file system. The second one is the planning data layout obtained by the *planner*. The *analyzer* first calculates the data access cost (T_O) for all file requests under the original data layout according to Equation (1). Then it obtains the data access cost (T_N) under the new (planning) data layout. Once these two costs are obtained, the replication benefit for the access pattern can be defined as follows:

$$B = T_O - T_N. \tag{5}$$

If *B* is larger than zero, then it is beneficial to carry out the actual replication with the planning data layout. Otherwise, serving the requests from the original layout helps I/O performance and there is no need to replicate. The higher performance benefit that the scheme can potentially achieve, the higher priority the access pattern should be replicated. In order to make the ultimate data replication decisions, the *analyzer* stores the replication benefits of all identified access patterns in a global pattern benefit table *PBT*, which will be lately used by the *replicator*.

4.5 Cost-Effective Data Replication

As the replica space is usually limited, the *replicator* ultimately carries out cost-effective data replication based on three factors: (1) the available free space in the parallel file system, indicating whether the system can accommodate the new replicas; (2) the performance benefit in the *PBT* table for an access pattern, indicating whether the I/O performance can be improved if it is replicated; (3) the rank of the performance benefit, indicating whether it incurs more performance benefit than other access patterns if it is chosen.

Algorithm 2 shows the data replication process for all access patterns. First, a global replica mapping table *RMT*, which keeps the data location information of the replicas, is initialized. The *RMT* is empty at the beginning, and will be continuously updated as new replicas are created in the replica space. For each access pattern, the algorithm checks if the requests fall into the replicas that have been created and allocated by consulting the *RMT*. If not, a new replica file will be created to hold the requests, and the location of the

RMT

…					
O_file	O_offset	R_file	R_offset	Length	R_flag
O_file	O_offset	R_file	R_offset	Length	R_flag
…					

Fig. 8. Structure of replica mapping table.

replica file is stored in the corresponding entry of the *RMT*. Supposing that there are c_{sys} free replica space and the data size of the current access pattern pat is c_p, then the algorithm will perform the actual data replication only when all the following conditions are true: (1) the system has enough free replica space, (2) the performance gain of the access pattern is positive, (3) the pattern belongs to the top-n unallocated patterns in the descending order of their performance gains. Otherwise, the algorithm will not carry out the data replication.

Algorithm 2. The Cost-Effective Data Replication Algorithm

```
1:  procedure CEDR($PBT, RMT$)
2:    $replica \leftarrow RMT\_lookup(pat)$
3:    if $reglica = NULL$ then
4:      $c_{sys} \leftarrow$ Data size of free replica space
5:      $c_p \leftarrow$ Data size of access pattern $pat$
6:      $Pat[n] \leftarrow$ top_n($\{x : x \in PBT \wedge x \notin RMT\}$)
7:      for each $pattern$ in $Pat[n]$ do
8:        if $c_p < c_{sys}$ and $pattern.benefit > 0$ and
          $pat = pattern$ then
9:          Create replica files
10:         for each data access in the $pat$ do
11:           Copy data from the original file to the replica
              files
12:           Add entries of replicas into $RMT$
13:         end for
14:       end if
15:     end for
16:   end if
17: end procedure
```

The actual data replication operations are complex because of overhead and consistency issues. To reduce the overhead, namely the interference with the normal I/O activities, the *replicator* is designed as a light-weight daemon that performs background data movements. It monitors a queue that contains all the selected access patterns that need to be replicated. Once the system has unused computing and storage resource, it will de-queue the queue and starts to replicate if the queue is not empty. After that, the *replicator* will read data from the original file and write them to the replica files according to the planning layout generated by the *planner*.

To maintain data consistency, the *RMT* table is used to keep data mapping information between original files and their replica files. As shown in Fig. 8, each entry of the *RMT* includes six fields. O_file and O_offset are the file name and offset for the data in the original file, R_file and R_offset are the file name and offset for the data in the replica file. Length is the size of requested data, and R_flag is a flag that indicates whether the replicated data is dirty. The R_flag is set when the data in the replica file is newly updated and it indicates that the data needs to be synchronized to the original file when the program ends. The *RMT* table is updated each time a new data block is copied to the replica files, ensuring that the up-to-date data is always maintained by the I/O system.

4.6 Data Redirection

The *redirector* is responsible for redirecting user's I/O requests to the original files or the replica files. Once redirected upon a file request, the *redirector* will look up the *RMT* to check if the request has a corresponding replica. Usually an application issues a request with three parameters: the identifier of the original file, the data offset, and the request size. If the corresponding entry is found, the *redirector* will obtain the new identifier, offset, and size in terms of the replica file, and send the request to the replica with the new request information. Otherwise, the request will be processed to the original file as usual.

The *redirector* resides in the MPI-IO library and runs only in the later runs of the applications. To redirect requests to the proper locations, applications need to be relinked to the library before its run without other modification.

5 Implementation

We implement CEDA within MPICH2 [18] on top of OrangeFS [6]. In the following, we discuss the primary challenges in the implementation.

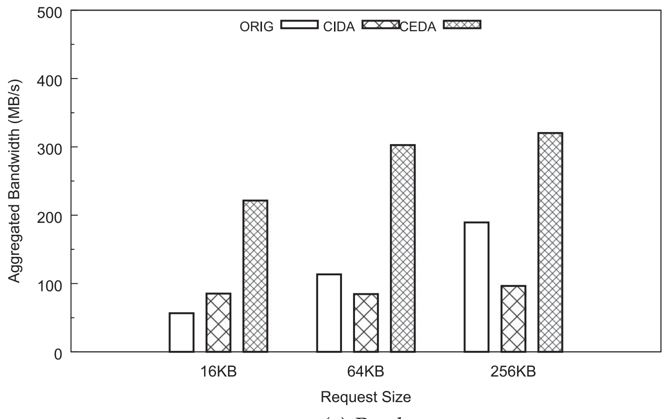
5.1 Replication Mapping Table

The *RMT* table is a critical structure to save the data mapping information between original files and replica files. As it is frequently accessed by the *redirector* and shared by multiple processes of the program, the effectiveness of *RMT* is a significant challenge. Inspired by the technique in the previous work [15], we use *Berkeley DB* [34] to implement *RMT* as a database file stored in the same directory as the MPI program. The *Berkeley DB* is configured as a hash table, and each record is a key-value pair. We generate mapID by encoding the following information: the program name, number of process, rank of the process, and the original file name. Each record in the hash table is a key-value pair; the key is the mapID and the value contains the data access information. By leveraging the advantage of the light-weighted database, the access contention and metadata operations are performed in an efficient way.

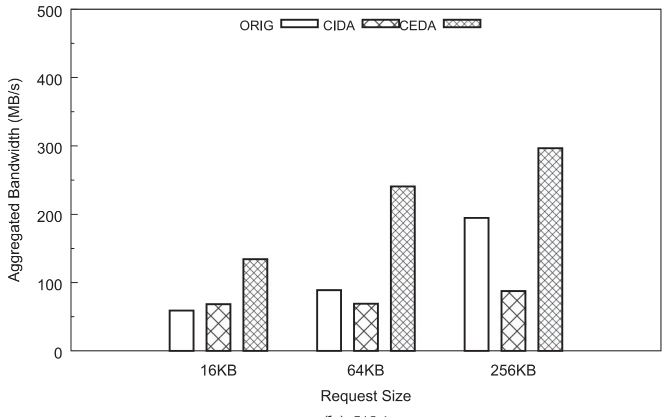
Furthermore, we use a list to maintain the most frequently accessed mapping entries to further reduce the in-memory mapping table size for efficient lookup. Changes to the mapping entries in memory are synchronously written to the storage in order to survive power failures. We modify the MPI library so that the mapping table is loaded with `MPI_Init()` and unloaded with `MPI_Finalize()`.

5.2 I/O Redirection Implementation

To redirect I/O requests to replica files with optimized access patterns, we modified the following standard MPI-IO functions:



(a) Read



(b) Write

Fig. 9. Bandwidths of IOR with various request sizes.

`MPI_File_open`: while opening a file, in addition to opening the original file, it also opens the corresponding replica files.

`MPI_File_read/write` (and other variants of read/write): for each I/O read or write operation, the function first uses the input parameters to check whether the request belongs to any of the replica file. If the requested data are found in *RMT*, the request will be redirected to the replica with new file name, offset. Otherwise, the request will then be served by the original file. All these operations are transparent to applications. In this way, the unused storage space can be intelligently utilized according to the I/O patterns.

`MPI_File_seek`: it calculates the offset and conducts the seek operation in the replica files.

`MPI_File_close`: it closes the opened replica files.

5.3 Other Data Consistency Issues

To maintain data consistency for the application, we adapt several approaches during the I/O redirection process. First, if the data of a write request can be found in the replica file according to the mapping table, then the request is re-directed to the replica file and the corresponding table entry is marked as dirty. Second, all read requests are redirected to their replicas, such that an obsolete original copy of the data will not be read within a program's run. Finally, we flush the data to the original files if the dirty flag in the entry is set when the application ends its execution. With these methods, the application can access the up-to-date data with higher I/O system performance.

6 Evaluation

6.1 Experimental Setup

We conduct experiments on a SUN Fire Linux cluster, where each node was configured with two Opteron quad-core processors, 8 GB memory and a 250 GB hard disk drive (HDD). Furthermore, four nodes are equipped with additional 100 GB SSD. All nodes are equipped with Gigabit Ethernet interconnection. The operating system is Ubuntu 13.04, the MPI library is MPICH2-1.4.1p1, and the parallel file system is OrangeFS 2.8.6.

We compare CEDA with the original I/O system (ORIG) where data replication is disabled. To ensure that the improvements only come from the cost-effective and distribution-aware replication, we also compare CEDA with a cost-ineffective but distribution-aware data replication scheme (CIDA) [15], which always selects non-contiguous file access patterns and replicates them contiguously in replica space with a specific data distribution. As CIDA only considers the data access costs of the replicas regardless of the costs of the original file requests, it can't guarantee the cost-benefit of the data replication and thus may experience inefficiencies in certain cases.

We use the popular benchmark IOR [20], HPIO [35], BTIO [27], and a real application [36] to test the I/O system performance.

We first share the experimental results for HDD-based parallel I/O systems, and next we show the results for SSD-based parallel I/O systems.

6.2 Evaluation on HDD-Based Parallel I/O Systems

By default we employ eight HDD-based nodes as file servers and eight other nodes as computing nodes. All file data are distributed on the eight servers with the default stripe size of 64 KB.

6.2.1 IOR Benchmark

IOR is a parallel file system benchmark developed at Lawrence Livermore National Laboratory [20]. It provides three APIs: MPI-IO, POSIX, and HDF5, we only use MPI-IO in the tests. During these benchmarks, IOR by default runs with 16 processes, each performing I/O operations on a shared file with request size of 64 KB.

Various Request Sizes. We first run IOR with various request sizes. Fig. 9 shows the I/O performance of IOR with request size from 16 to 256 KB. Each process issues random strided I/O requests and accesses data of 128 MB. We observe that CEDA outperforms ORIG and CIDA. By using the cost-effective distribution-aware data replication, CEDA improves read performance from 69.1 to 291.7 percent, and write performance from 57.3 to 171.2 percent respectively at request size 16, 64 and 256 KB, in comparison with ORIG. Compared with CIDA, CEDA also has better performance: the read performance is increased up to 257.8 percent, and write performance is increased as much as 249.7 percent. As the request size increases, IOR's bandwidth becomes higher because the increasingly amortized disk seek time reduces the penalty of non-sequential disk access. We also find CIDA to be only efficient to improve the original I/O system performance for small request size of 16 KB. For request size of 64 and 256 KB, CIDA even decreases the original I/O

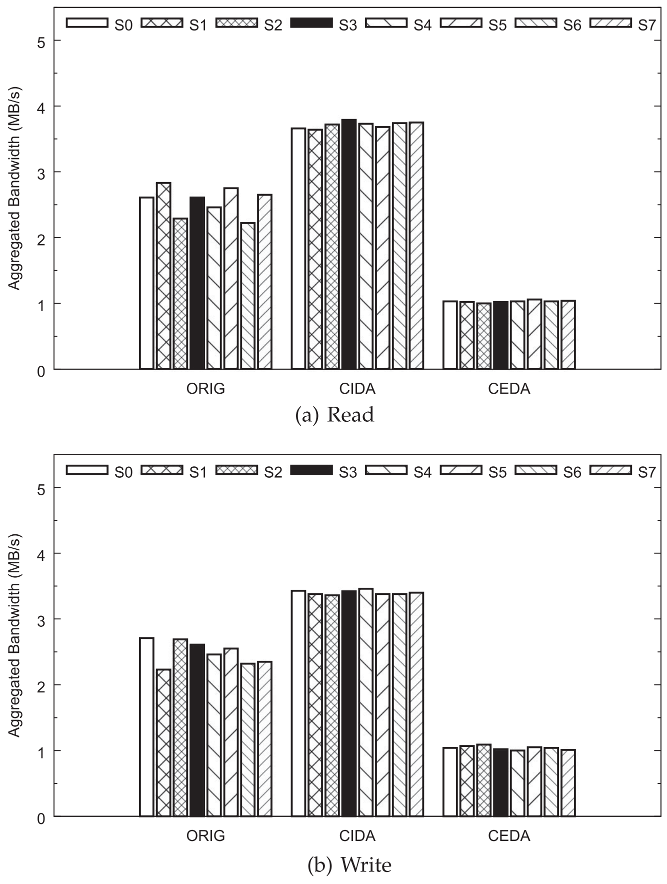


Fig. 10. I/O time on each file server different data replication schemes. S0-S7 refer to the eight servers.

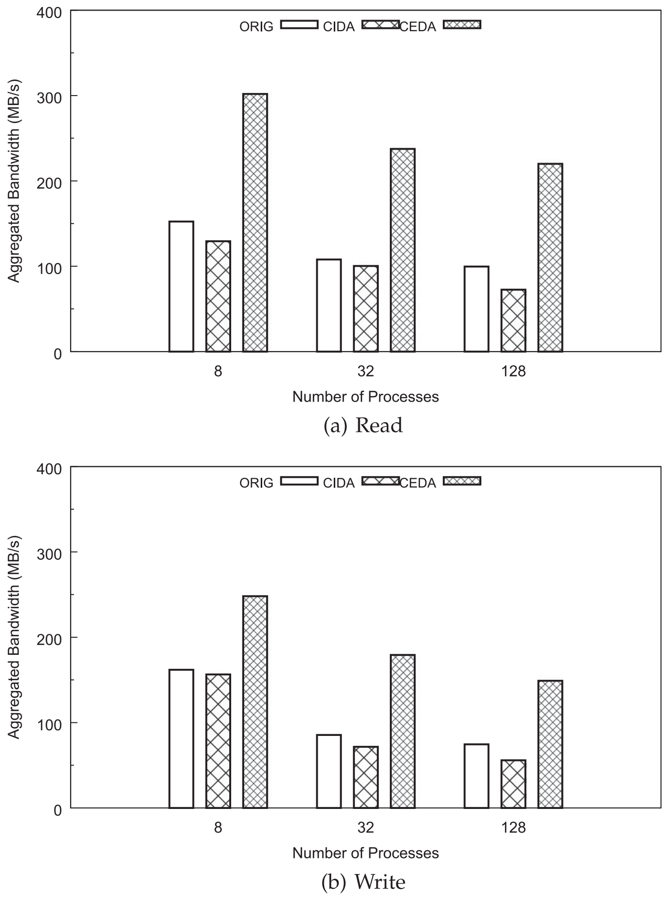


Fig. 11. Bandwidths of IOR with various process numbers.

performance by up to 49.1 percent. This is because combining non-contiguous small file requests may lead to more contiguous accesses on disk but it is not the case for larger non-contiguous file requests, as shown in Section 2. On the other hand, CEDA always shows superior I/O performance.

To give a detailed explanation for CEDA's performance improvement, Fig. 10 plots the I/O time of each server when IOR issues strided requests at the size of 64 KB. The I/O time refers to the accumulated sub-request completion time on a server, which is normalized to the minimal time on all servers under the CEDA layout. We observe that the I/O loads are skewed across servers under ORIG because IOR issues non-uniform sub-requests to servers. In contrast, CIDA and CEDA have nearly even loads because they place the same amount of data on each server in the data replication process. However, CEDA has less I/O time. This is because it only makes cost-effective data replication by making each disk to serve more contiguous sub-requests, while CIDA blindly replicates all non-contiguous file requests even it may decrease I/O performance.

Various Process Numbers. We then evaluate IOR with various process numbers. The IOR benchmark is executed with 8, 32, and 128 processes at a fixed request size of 64 KB. As shown in Fig. 11, the results are similar to the previous test: CEDA improves I/O performance for both read and write requests over CIDA and ORIG. Compared with ORIG, the I/O bandwidth is increased 98.1, 119.9, and 121.7 percent respectively for reads in terms of various numbers of processes, and 53.4, 109.7, and 99.8 percent for writes. In contrast to CIDA, the read performance achieves a 133.4, 136.9, and 202.9 percent improvement, and write performance achieves a 58.7, 150.3, and 167.1 percent improvement. As the number of processes increases, the I/O bandwidth gets lower because each server needs to serve more processes' requests and the competition among processes gets more severe. However the rate of performance degrading of CEDA is not as substantial as those of ORIG and CIDA because it makes each server handle less processes. These results illustrate that CEDA a high scalability in terms of the number of processes.

Various Server Numbers. We next examine the I/O performance with various server numbers. The number of servers is varied from two to eight, and the request size is kept to 64 KB. As the Fig. 12 depicts, CEDA improves I/O performance for both data reads and writes: read performance increases from 18.5 to 166.9 percent, while write performance improves from 62.9 to 171.2 percent in comparison with ORIG. Compared with CIDA, CEDA achieves an improvement up to 257.8 percent for reads and 248.5 percent for writes. In the experiments, read and write performance improve as the number of servers increases. This is because the requests can be distributed on more servers, which contribute to the overall I/O performance by utilizing a higher I/O concurrency. We also note that, with larger number of servers, the performance improvement of CEDA over CIDA is more significant. This is because CEDA improves the data access continuity in a single server and increases the load balance among multiple servers.

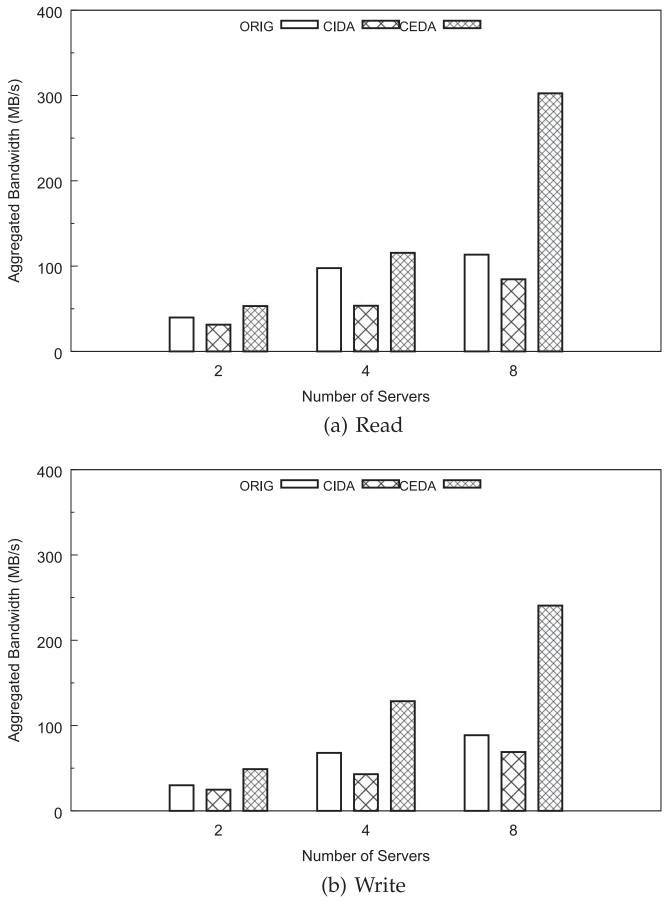


Fig. 12. Bandwidths of IOR with various server numbers.

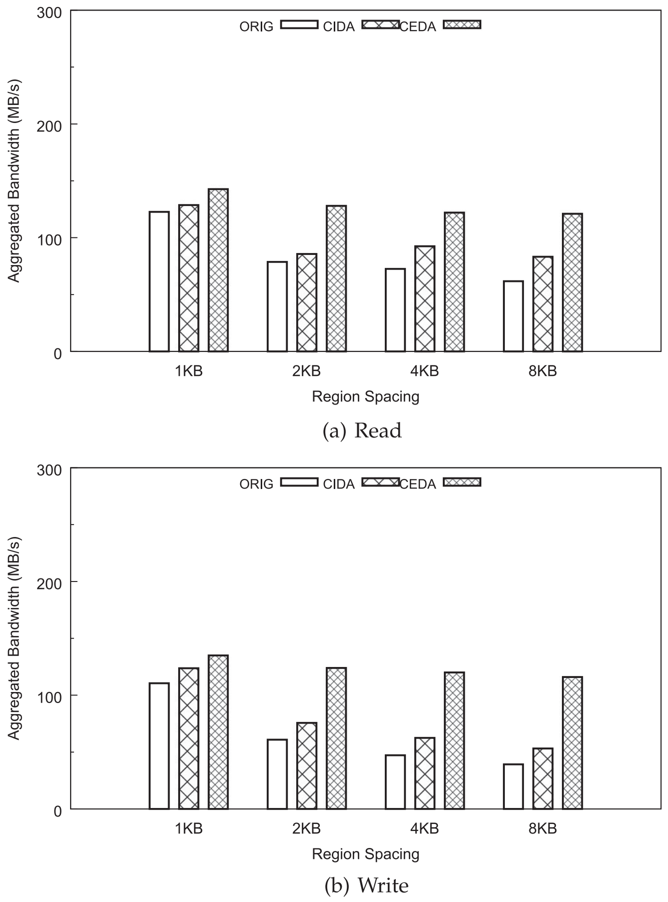


Fig. 13. Bandwidths of HPIO with various region spacings.

[*Summary*] The above study shows that CEDA can improve IOR performance in terms of different request sizes, numbers of processes, and numbers of servers. With the increase of the request size, the number of processes, and the number of the servers, CEDA can obtain more significant performance improvements over existing replication schemes.

6.2.2 HPIO Benchmark

HPIO is a program designed by Northwestern University and Sandia National Laboratories to systematically evaluate parallel I/O system performance [35]. This benchmark can generate various data access patterns by changing three parameters: region count, region spacing, and region size. The region spacing is used to generate non-contiguous data access patterns. In our experiment, the number of process is 32, the region count is 4,096, the region size is 16 KB, and the region spacing is varied from 1 to 8 KB.

Fig. 13a shows the I/O bandwidth for read requests. Compared to ORIG and CIDA, CEDA can increase the overall I/O bandwidth. CEDA can increase the I/O bandwidth over CIDA by 10.9, 49.3, 31.8, and 45.4 percent respectively for reads. It means that CEDA is effective with respect to HPIO benchmark. We also note that, as the region spacing increases, the performance speedup gets more obvious. This is because non-contiguous I/O requests can benefit more from the physical continuity-aware data replication. Although the I/O access of each process is non-contiguous, it is not as random as the IOR benchmark, thus the improvements for HPIO are not as significant as those for IOR. This also confirms the adaptability of CEDA: when the application's I/O accesses have a poorer bandwidth, more benefit is gained by using CEDA. For write operations, the performance shows a similar trend as presented in Fig. 13b.

6.2.3 BTIO Benchmark

BTIO represents a typical scientific application with interleaved intensive computation and read/write mixed I/O phases [27]. BTIO uses a Block-Tridiagonal (BT) partitioning pattern to solve the three-dimensional compressible Navier-Stokes equations. We consider the Class *A* and *simple* subtype workload in the experiments. In this case, BTIO writes and reads a total size of 1.69 GB data with individual I/O requests (non-collective I/O). We use 4, 16, and 64 compute processes since BTIO requires a square number of processes. Output file is striped across four servers.

Fig. 14 plots the aggregate I/O bandwidths. Compared with the original I/O system (ORIG), CEDA achieves 37.1, 41.3, and 54.8 percent improvement with 4, 16, 64 processes, respectively. In contrast to CIDA, CEDA also demonstrates performance advantages.

6.2.4 Real Application

Finally we evaluate the performance of CEDA with a real application's I/O trace [36]. We choose this application because it has mixed access patterns which can show the adaptivity of our scheme. In the application, each process

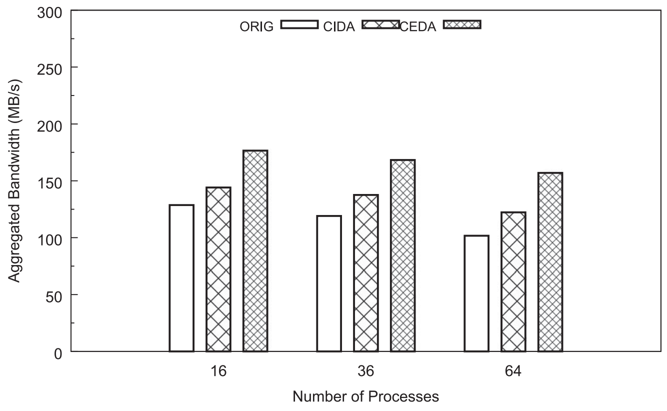


Fig. 14. Bandwidths of BTIO benchmark.

sends I/O requests in a non-uniform way at different parts of a shared file. In summary, the file can be roughly divided into multiple segments, in which each process issues requests with different sizes iteratively. We replay the data accesses of the application according to the I/O trace. In the experiment, we employ eight nodes as computing nodes, and four nodes as file servers. As shown in Fig. 15, we find that CEDA obtains 85.1 and 63.5 percent performance improvement compared to ORIG and CIDA respectively. The results indicate that CEDA is effective for applications with complex I/O access patterns.

6.3 Evaluation on SSD-Based Parallel I/O Systems

We select eight nodes as computing nodes and configure the file system on four SSD-based file servers. The stripe size of the file system is 64 KB. We run IOR with 32 processes, each with random I/O requests to access 256 MB of data.

Fig. 16 shows the I/O performance of IOR with various request sizes on the SSD-based I/O system. The results are similar to the HDD-based case: CEDA has the best performance among the three schemes. As it can be observed from the figures, when compared to ORIG, CEDA improves read performance up to 39.8 percent, and write performance up to 72.8 percent for request sizes of 4, 16 and 64 KB. In contrast to CIDA, CEDA improves read performance up to 7.5 percent, and write performance up to 14.7 percent. Another observation is that the performance improvements in SSD-based case are not as significant as those in the HDD-based case. This is because SSD is less sensitive to the sequentiality of the requests. In other words, combining non-contiguous small requests into contiguous ones on an

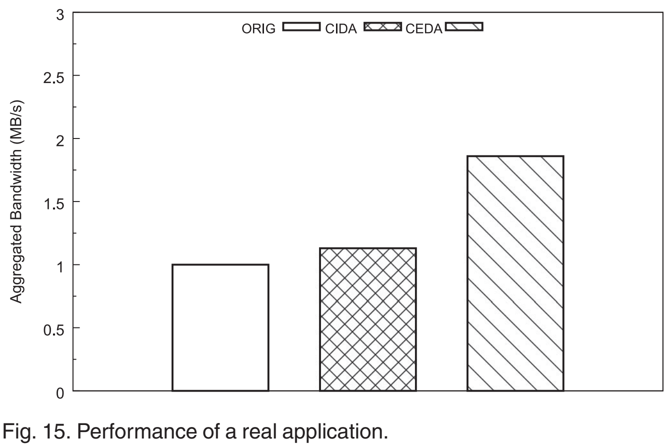


Fig. 15. Performance of a real application.

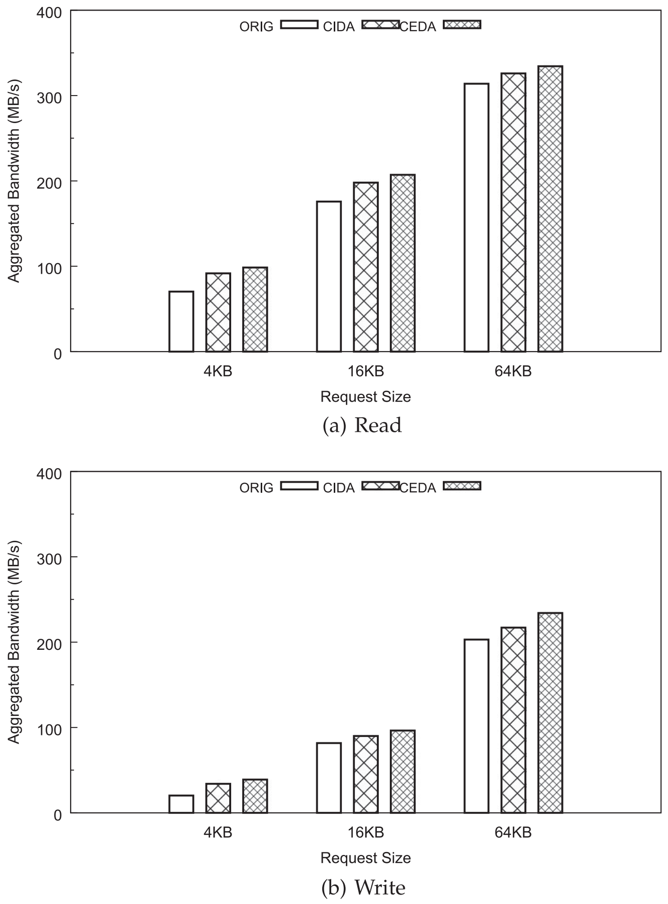


Fig. 16. Bandwidths of IOR with various request sizes.

HDD can largely improve the I/O efficiency of HDD, but it has less impact on the SSD performance. However, CEDA always brings performance benefits over ORIG and CIDA for SSD-based I/O systems. These results show that CEDA also applies to I/O systems with SSD-based file servers.

6.4 Overhead Analysis

While the gains due to data replication are promising, CEDA incurs resource overhead.

6.4.1 Performance Overhead

In the "profiling phase", the profiler uses IOSIG to collect I/O traces during the application's first run. Previous work shows that IOSIG only incurs very low overhead in tracing I/O accesses [33]. In the following two phases, since the pattern analysis and planning are carried out off-line and only once, the CPU and memory overhead is also acceptable for most HPC computing systems.

In the "redirection phase", the redirector needs to determine where to send the requests. It is necessary to evaluate the following two possible sources of overhead during runtime: 1) During file open operation, the *redirector* needs to search the access pattern in the Pattern Database; 2) During file read/write operation, the *Redirector* needs to check if the opened files contain the requested data, thus to determine the proper locations to send the requests.

To show the redirection overhead, we run IOR with various request sizes from 4 to 64 KB. The process numbers are 16, 32, and 64. The file system is built on eight HDD-based

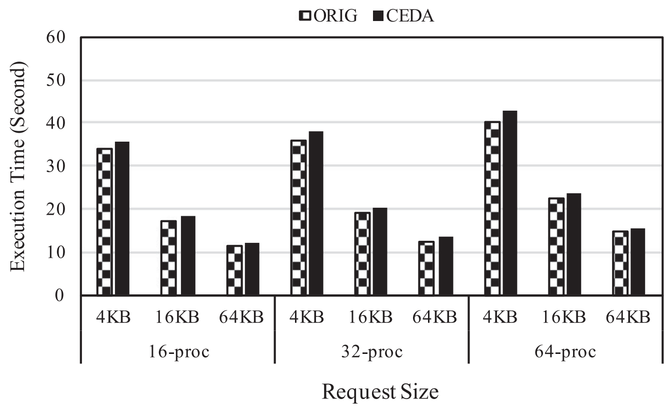


Fig. 17. CEDA performance overhead.

file servers and each process writes a 1,024 MB of data with a contiguous access pattern. We intentionally do not add the patterns in the database so that the system will make no data replication. This causes *redirector* to redirect all requests to the original I/O system. Fig. 17 shows that the introduced overhead is acceptable.

6.4.2 Metadata Space Overhead

To maintain data consistency, CEDA stores the replica mapping information in the replica remapping table (RMT) in the same parallel file system, incurring additional storage space.

The system has a maximal space overhead when all the requests are of 4 KB. Assuming that the available storage space for data replication is S GB and that each entry of RMT in our implementation occupies $6 * 4$ B, the maximal number of records in *RMT* then is $S/4 * 10^6$. Therefore, the maximal metadata space overhead is 0.6 percent of the replica space, an acceptable requirement for a storage cluster.

7 Related Work

In the past years several efforts have been devoted to data replication. BORG [13] automatically copies selected data blocks to a dedicated disk partition, such that the reorganized disk layout accommodates the I/O access patterns. FS2 [14] dynamically places data with multiple additional copies in free blocks of a file system, such that the nearest replica can be accessed to benefit disk I/O operations. Similarly, Koller et al. [37] propose to direct read requests to the intrinsic duplication of a storage system or the additional replicas to reduce I/O time. However, these techniques are designed for a serial I/O system with a *single* disk. Instead, CEDA are designed for a parallel I/O system with *multiple* disks.

Data replication has also been widely used in parallel I/O systems. Hybrid Replication [21] stores each file with three copies such that each file request can be redirected to the proper replica with the smallest access cost. HDFS [22] and GPFS-SNC [38] create multiple copies for each file in different physical locations, such that I/O requests can be redirected to the nearest location to reduce access costs. InterferenceRemoval [12] replicates file segments to designated servers, such that requests are re-directed to other servers to reduce interference on each node. Wang et al. [30] propose to replicate frequently accessed data chunks to the disks of compute nodes to reduce access latency. Scarlett [43] uses popularity-based data replication schemes in HDFS to improve the performance of Hadoop clusters. Compared to these methods, CEDA carries out pattern-based replication, which can further save storage space because it only replicates the accessed data.

PDLA [15] replicates identified data access patterns, and distributes them on multiple servers with optimized data layouts based on data placement cost analysis. RADAR [16] presents a partial data replication system, which replicates access patterns using the object abstraction in a parallel file system. Recent work [17] selects more general access patterns and replicates the data of interest in multiple reorganized layouts. However, all these studies make replication decisions based on file access information regardless of the performance benefits obtained by the replication; that is, they always select non-contiguous file requests and continuously place them in the replica file(s). In contrast, CEDA makes replication decisions with physical data access information on disk by considering data distribution on servers and it only carries out cost-effective data replication based on replication benefit analysis.

For data reorganization, PLFS [29] stores write data to an original logical file in a set of new created log-structured physical files. Although the write performance can be largely enhanced, the read performance from multiple physical files may be a concern due to the inevitable data restructuring. However, CEDA handles both read and write patterns to improve I/O performance. Furthermore, the data reorganization in PLFS is based on the logical file access information, which also owns the similar drawbacks of traditional data replication schemes. As opposed to PLFS, CEDA makes cost-effective data reorganization by considering physical data distribution on multiple disks of the servers.

Besides I/O performance, data replication has also been used to boost system reliability. Several major parallel file systems, such as Lustre [7] and GPFS [8], provide built-in data replication for enhanced fault tolerance. HDFS [22] uses multiple replicas to serve the same goal. Wang et al. [39] propose to schedule a job to replicated files on active nodes, such that re-execution of the job can be avoided. Similar redundant data placement approaches [40], [41] are used to ensure system availability when some storage devices enter or leave the system. As opposed to these approaches, CEDA aims to improve I/O performance instead of availability.

Currently, several MapReduce applications need to analyze the raw data obtained from HPC applications. To make datasets generated by HPC applications more accessible for data analytics applications, MRAP [42] reorganizes datasets according to access patterns when copying them from HPC storage to MapReduce storage. Our work differs from these efforts for which we focus on I/O optimization in general purpose parallel file systems, such as PVFS2 [32] and OrangeFS [6].

8 Conclusions

In this study, we have proposed CEDA, a cost-effective distribution-aware data replication scheme to improve parallel I/O system performance. Unlike data replication schemes that consider logical file access information and assume that contiguous file requests are more efficient than non-contiguous ones, CEDA considers physical data accesses on multiple servers and makes cost-effective data replication. CEDA leverages a cost model to evaluate the I/O access time of a file request with a given data layout, and only performs the actual data replication when the replication can achieve performance benefits. We have implemented CEDA in the MPICH I/O library on top of the OrangeFS parallel file system. Experimental results with representative benchmarks and a real application show that CEDA is a viable solution to improve parallel I/O system performance.

In future work, we plan to evaluate CEDA on a large-scale cluster with other applications, to gain more insights into its performance behaviors. We also intend to exploit the transient access patterns of applications to optimize parallel I/O system performance.

Acknowledgments

The authors are thankful to Tao Li from Hunan University for his constructive help toward this work. This study was supported in part by the National Science Foundation of China under Grant No. 61572377.

References

[1] R. Latham, R. Ross, B. Welch, and K. Antypas, "Parallel I/O in Practice," in *Tutorial of SC15*, 2015.

[2] Y. Kim, S. Atchley, G. R. Valle, and G. M. Shipman, "LADS: Optimizing data transfers using layout-aware data scheduling," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 67–80.

[3] S. He, X.-H. Sun, and B. Feng, "S4D-Cache: Smart selective SSD cache for parallel I/O systems," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 514–523.

[4] 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 *Proc. 44th Int. Conf. Parallel Process.*, 2015, pp. 340–349.

[5] S. He, X.-H. Sun, and A. Haider, "HAS: Heterogeneity-aware selective data layout scheme for parallel file systems on hybrid servers," in *Proc. 29th IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 613–622.

[6] "Orange File System," 2018. [Online]. Available: http://www.orangefs.org/

[7] S. Microsystems, "Lustre file system: High-performance storage Architecture and Scalable Cluster File System," Tech. Rep. Lustre File System White Paper, 2007.

[8] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, pp. 231–244.

[9] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Proc. 7th Symp. Frontiers Massively Parallel Comput.*, 1999, pp. 182–189.

[10] S. He, Y. Wang, X.-H. Sun, C. Huang, and C. Xu, "Heterogeneity-aware collective I/O for parallel I/O systems with hybrid HDD/SSD servers," *IEEE Trans. Comput.*, vol. 66, no. 6, pp. 1091–1098, Jun. 2017.

[11] S. He, Y. Wang, X.-H. Sun, and C. Xu, "HARL: Optimizing parallel file systems with heterogeneity-aware region-level data layout," *IEEE Trans. Comput.*, vol. 66, no. 6, pp. 1048–1060, Jun. 2017.

[12] X. Zhang and S. Jiang, "InterferenceRemoval: Removing interference of disk access for MPI programs through data replication," in *Proc. 24th ACM Int. Conf. Supercomputing*, 2010, pp. 223–232.

[13] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reorganization for self-optimizing storage systems," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 183–196.

[14] H. Huang, W. Hung, and K. G. Shin, "FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption," in *Proc. 20th ACM Symp. Operating Syst. Principles*, 2005, pp. 263–276.

[15] 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 *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 345–356.

[16] J. Jenkins, X. Zou, H. Tang, D. Kimpe, R. Ross, and N. F. Samatova, "RADAR: Runtime asymmetric data-access driven scientific data replication," in *Proc. Int. Supercomputing Conf.*, 2014, pp. 296–313.

[17] 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 *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 356–365.

[18] A. N. Lab, "MPICH2: A high performance and widely portable implementation of MPI," 2018. [Online]. Available: http://www.mcs.anl.gov/research/project-detail.php?id=2

[19] J. He, H. Song, X.-H. Sun, Y. Yin, and R. Thakur, "Pattern-aware file reorganization in MPI-IO," in *Proc. 6th Workshop Parallel Data Storage*, 2011, pp. 43–48.

[20] "Interleaved Or Random (IOR) Benchmarks," 2018. [Online]. Available: http://sourceforge.net/projects/ior-sio/

[21] H. Song, Y. Yin, Y. Chen, and X.-H. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 37–48.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[23] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application I/O signatures from noisy server-side traces," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 213–228.

[24] S. He, X.-H. Sun, B. Feng, X. Huang, and K. Feng, "A cost-aware region-level data placement scheme for hybrid parallel I/O systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–8.

[25] S. He, Y. Wang, and X.-H. Sun, "Improving performance of parallel I/O systems through selective and layout-aware SSD cache," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 2940–2952, Oct. 2016.

[26] S. He, Y. Wang, and X.-H. Sun, "Boosting parallel file system performance with heterogeneity-aware selective data layout," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2492–2505, Sep. 2016.

[27] "The NAS parallel benchmarks," 2018. [Online]. Available: www.nas.nasa.gov/publications/npb.html

[28] T. M. Madhyastha and D. Reed, "Exploiting global input output access pattern classification," in *Proc. ACM/IEEE Conf. Supercomputing*, 1997, pp. 9–9.

[29] 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 *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, pp. 1–12.

[30] Y. Wang and D. Kaeli, "Profile-guided I/O partitioning," in *Proc. 17th Annu. Int. Conf. Supercomputing*, 2003, pp. 252–260.

[31] 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 *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2008, pp. 1–12.

[32] P. H. Carns, I. WalterB. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel virtual file system for linux clusters," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 317–327.

[33] Y. Yin, S. Byna, H. Song, X.-H. Sun, and R. Thakur, "Boosting application-specific parallel I/O optimization using IOSIG," in *Proc. 12th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2012, pp. 196–203.

[34] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley DB," in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 183–191.

[35] A. Ching, A. Choudhary, W.-K. Liao, L. Ward, and N. Pundit, "Evaluating I/O characteristics and methods for storing structured scientific data," in *Proc. 20th Int. Parallel Distrib. Process. Symp.*, 2006, Art. no. 10.

[36] "Application I/O traces: Anonymous LANL App2," 2014. [Online]. Available: http://institutes.lanl.gov/plfs/maps/

[37] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," in *Proc. 8th USENIX Conf. File Storage Technol.*, pp. 211–224, 2010.

[38] K. Gupta, R. Jain, I. Koltsidas, H. Pucha, P. Sarkar, M. Seaman, andD. Subhraveti, "GPFS-SNC: An enterprise storage framework for virtual-machine clouds," *IBM J. Res. Develop.*, vol. 55, no. 6, pp. 1–10, 2011.

[39] C. Wang, Z. Zhang, X. Ma, S. S. Vazhkudai, and F. Mueller, "Improving the availability of supercomputer job input data using temporal replication," *Comput. Sci. - Res. Develop.*, vol. 23, pp. 149–157, 2009.

[40] A. Brinkmann, S. Effert, F. M. auf der Heide, and C. Scheideler, "Dynamic and redundant data placement," in *Proc. 27th Int. Conf. Distrib. Comput. Syst.*, 2007, pp. 29–38.
[41] A. Brinkmann and S. Effert, "Redundant data placement strategies for cluster storage environments," in *Proc. 12th Int. Conf. Principles Distrib. Syst.*, 2008, pp. 551–554.
[42] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "MRAP: A novel MapReduce-based framework to support HPC analytics applications with access patterns," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 107–118.
[43] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris, "Scarlett: Coping with skewed content popularity in MapReduce clusters," in *Proc. Sixth Euro. Conf. Comput. Syst.*, 2011, pp. 287–300.

Shuibing He received the PhD degree in computer science and technology from the Huazhong University of Science and Technology, China, in 2009. He is now an associate professor with the School of Computer Science, Wuhan University, China. His research areas include file and storage systems, high-performance computing, and distributed computing. He has published around 50 papers in the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *ACM Transactions on Autonomous and Adaptive Systems*, the *ACM Transactions on Embedded Computing Systems*, ICDCS, IPDPS, ICPP, CLUSTER, and HiPC.

Xian-He Sun received the BS degree in mathematics from Beijing Normal University, China, in 1982, and the MS and PhD degrees in computer science from Michigan State University, in 1987 and 1990, respectively. He is a distinguished professor with the Department of Computer Science, the Illinois Institute of Technology (IIT), Chicago. His research interests include parallel and distributed processing, memory and I/O systems. He is a fellow of the IEEE.