HARL: Optimizing Parallel File Systems with Heterogeneity-Aware Region-Level Data Layout

Shuibing He, Yang Wang, Xian-He Sun, *Fellow, IEEE*, and Chengzhong Xu, *Fellow, IEEE*

Abstract—Parallel file system (PFS) is commonly used in high-end computing systems. With the emergence of solid state drives (SSDs), hybrid PFS, which consists of both HDD and SSD servers, provides a practical I/O system solution for data-intensive applications. However, most existing data layout schemes are inefficient for hybrid PFS due to their unawareness of server heterogeneities and workload changes in different parts of a file. In this study, we propose a heterogeneity-aware region-level data layout scheme, HARL, to improve the data distribution of a hybrid PFS. HARL first divides a file into fine-grained, varying sized regions according to the workload features of an application, then determines appropriate file stripe sizes on servers for each region based on the performance of heterogeneous servers. Furthermore, to further improve the performance of a hybrid PFS, we propose a dynamic region-level layout scheme, HARL-D, which creates multiple replicas for each region and redirects file requests to the proper replicas with the lowest access costs at the runtime. Experimental results of representative benchmarks and a real application show that HARL can greatly improve I/O system performance, and demonstrate the advantages of HARL-D over HARL.

Index Terms—Parallel I/O system, parallel file system, solid state drive, data layout, hybrid parallel file system

1 Introduction

Many large-scale applications are becoming data-intensive, and I/O performance is turned out to be the bottleneck of computer systems. To tackle this challenge, parallel file systems (PFSs), such as OrangeFS [1], Lustre [2], GPFS [3] and PanFS [4], is often used to form the base of high-performance computer systems. By serving a client request concurrently from multiple file servers, PFSs can dramatically improve the aggregate I/O bandwidth of underlying storage systems. However, a frustrating aspect of PFSs is that their common-case performance is often worse than their reported peak performance [5], [6].

The new storage technologies, such as flash-based solid state drives (SSD), provide a possible alternative solution for I/O system design. Unlike traditional HDDs, SSDs are composed of semiconductor chips, and thus provide higher I/O performance [7]. Although having performance advantage over HDDs, SSDs bring cost concerns when they are used completely to replace HDDs in a large cluster. Thus, a hybrid PFS, which consists of both HDD servers (HServer) and SSD servers (SServer), is more practical for HPC systems under a limited storage budget [5], [8].

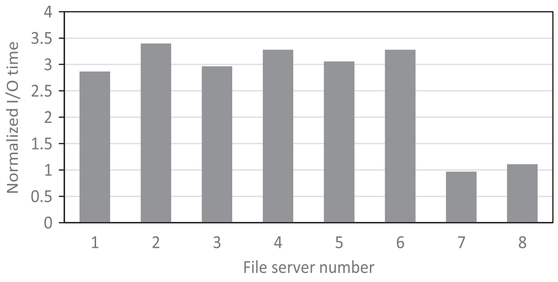
Although hybrid PFSs are promising, their efficiency relies on an efficient data layout scheme, an algorithm defining how a file's data is distributed on available storage servers. Currently, most existing layout schemes distribute all file data across multiple servers with a fixed-size stripe [6], as shown in Fig. 2a. This can provide concurrent data access from multiple servers and come with even data placement on each server. While these schemes are widely used and simple to implement, they are typically designed for PFSs with homogeneous servers. When applied to hybrid PFSs, these schemes would raise the following challenges.

First, the performance gap between HServers and SServers can significantly degrade the performance of PFSs. SServers always have higher performance than HServers, thus they usually require less I/O time to complete the same amount of data accesses. However, current layout schemes generally assign identical stripes to both HServers and SServers, leading to severe load imbalance among heterogeneous servers. To illustrate this issue, we ran IOR [9] with 512 KB request size and 16 processes on a hybrid OrangeFS file system with the default layout (Stripe size is 64 KB). Fig. 1a shows the I/O time on each server, normalized to the minimum of all servers. We can observe that the slow HServers (Server 1-6) take roughly 300 percent I/O time compared with fast SServers (Server 7-8), which means that the potential of the high-performance SServers are not fully underutilized.

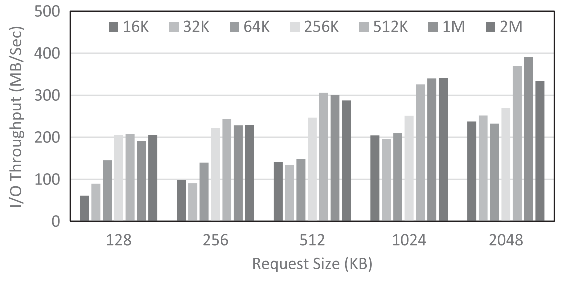
Second, complex I/O patterns may also compromise the efficiency of I/O systems. Current layout schemes often adopt a fixed-size stripe for the whole file [10], however the I/O patterns of different parts of a file can be totally different [10], [11]: request sizes can be large at one file chunk but small at another; request types can be read operation in one I/O phase but write in another. As a data layout is only efficient for a certain type of workloads, such file-level static striping methods may not adapt to the workload changes. Fig. 1b shows the performance of IOR with varied request sizes from 128 to 2,048 KB under fixed stripe sizes from 16 KB

- *S. He is with the State Key Laboratory of Software Engineering, Computer School, Wuhan University, Luojiashan, Wuhan, Hubei 430072, China. E-mail: heshuibing@whu.edu.cn.*
- *Y. Wang and C. Xu are with Shenzhen Institute of Advanced Technology, Chinese Academy of Science Xueyuan Blvd. 1068, Shenzhen 518055, China. E-mail: {yang.wang1, cz.xu}@siat.ac.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 4 Sept. 2016; revised 21 Nov. 2016; accepted 6 Dec. 2016. Date of publication 8 Dec. 2016; date of current version 16 May 2017.
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.2016.2637905



(a) I/O time of each server under a fixed I/O pattern and stripe size



(b) Throughput with varied I/O patterns and stripe sizes

Fig. 1. Performance statistics of IOR in a hybrid PFS. In (a), server 1-6 are HServers, and server 7-8 are SServers. In (b), the legend "#K" denotes the data layout with a fixed-size stripe of #KB on each server.

to 2 MB. We can see that there is a huge variation in I/O bandwidth under different I/O workloads and stripe sizes.

In this paper, we propose a heterogeneity-aware region-level (HARL) data layout scheme, named HARL, to address the challenges in the current data distribution of PFSs. Since a fixed-size stripe is sub-optimal for either heterogeneous servers in the storage system or applications with complex I/O patterns, HARL relies on a storage and application-aware allocation scheme to determine the optimal file stripe sizes on heterogeneous servers. More specifically, HARL first divides a file into fine-grained regions according to the changes of application's I/O workload; then, HARL assigns appropriate file stripe sizes to both HDD and SSD servers based on their storage performance for each file region. It essentially represents a promotion from the traditional one dimensional fixed-size stripe layout to a two-dimensional varied-size stripe layout. In this way, HARL can significantly speeds up I/O system performance by mitigating load imbalance among heterogeneous servers and increasing I/O efficiency of data accesses in each file region.

Since a static data layout scheme is not the most efficient way to serve varying data accesses, we propose HARL-D, a dynamic data layout scheme that leverages data replication to further improve the performance of a hybrid PFS. For each file region, HARL-D creates multiple replicas, each with optimized stripe sizes on HServers and SServers. By redirecting each file request to the most appropriate replica, HARL-D can further improve the overall I/O performance. As opposed to the static data layout scheme heterogeneity-aware region-level in our conference version [12], whose stripe sizes are immutable after the initial creation, such a dynamic policy is more flexible to adapt to the varied data accesses at runtime.

Notably, HARL is transparent to applications, as such it requires no modifications to the applications and can be integrated with any hybrid PFS in a simple way. In summary, this study makes the following contributions.

- A mathematical cost model, which considers I/O patterns, system architecture, network overhead, storage performance and data layout characteristics, is introduced to evaluate the data access time of one file request in a hybrid PFS.
- A static region-level data layout scheme (HARL), which logically divides a file into regions and then optimizes the stripe sizes on HServers and SServers for each region based on the cost model, is presented to optimize the performance of a hybrid PFS.
- A dynamic region-level data layout scheme (HARL-D), which creates multiple replicas for each file region and redirects file requests to the preferable replica with the lowest access cost, is described to further improve the performance of a hybrid PFS.
- A prototype of the proposed data layout scheme is implemented and integrated into MPICH2 [13]. Experimental results with representative benchmarks and an application show that HARL can significantly improve the I/O throughput of a hybrid PFS, and demonstrate the advantages of HARL-D over HARL.

The reminder of this paper is organized as follows. Section 2 discusses the related work. The static and dynamic region-level data layout scheme are described in Sections 3 and 4. Section 5 presents the performance evaluation with commonly used benchmarks. Finally, the conclusions are summarized in Section 6.

2 Related Work

In this section we briefly discuss some related work on improving the performance of parallel I/O systems from three aspects.

I/O Access Reorganization. A great deal of research has focused on reorganizing I/O accesses at the parallel I/O middleware layer. For example, instead of accessing multiple small, noncontiguous requests, data sieving [14] applies the strategy of accessing a contiguous chunk created by gathering the noncontiguous requests. Datatype I/O [15] and List I/O techniques [16] allow noncontiguous I/O requests to be converted into a single I/O request, thereby limiting the number of total requests. Collective I/O [14] also optimizes I/O performance by rearranging I/O accesses into a larger contiguous request, but it considers multiple processes of a parallel program instead of an individual process. Two-phase I/O [17] is one of the implementations of collective I/O operations. It consists of two main phases: shuffle phase and I/O phase. For write optimization, PLFS [18] redirects multiple parallel requests to a set of efficiently reorganized log-formatted files to generate more sequential write requests, but the read performance of these files may not be ideal due to the inevitable data restructuring.

Data Layout in HDD-Based File Systems. Parallel file systems support different data layout strategies, which allow for numerous data layout optimization methods. Several techniques, including data partition [19], [20], data

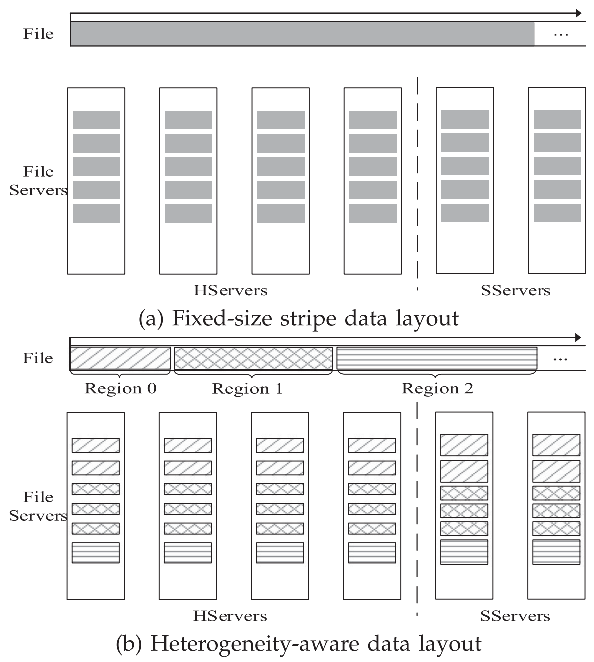

File
File Servers
HServers
SServers
(a) Fixed-size stripe data layout
Region 0
Region 1
Region 2
(b) Heterogeneity-aware data layout

Fig. 2. *Two data layout schemes in a hybrid parallel file system.* This figure shows how a file's data are distributed on HServers and SServers, focusing on the stripe size configuration. The height of the rectangle on each server represents the stripe size assigned to them. While case (a) uses a fixed-size stripe for each server within the whole file, case (b) divides a file into multiple regions and uses varied-size stripes for HServers and SServers to distribute data in each region.

migration [21], and data replication [6], [22], are applied to optimize data layouts depending on I/O workloads. Segment-level layout scheme logically divides a file to several parts and appoints an optimal stripe size for each part [10]. However, it only considers application heterogeneity, and thus it could be potentially used in conjunction with our proposed solution. Another methodology, server-level adaptive layout strategy, selects different stripe sizes depending upon the type of the file server [23]. PARLO is designed for accelerating queries on scientific datasets by applying user specified optimizations [24]. AdaptRaid confronts load imbalance in heterogeneous disk arrays [25] using an adaptive number of blocks, which cannot be implemented in PFSs.

Data Layout in SSD-Based File Systems. SSDs are commonly integrated into parallel file systems due to their performance benefits. A popular method is to use SSDs as a cache of traditional HDDs, e.g., Sievestore [26] and iBridge [27]. Another widely used approach is to utilize SSDs as a part of data storage, such as I-CASH [28] and Hystor [29]. Wu et al. [30] discusses the data placement and scheduling trade-offs for hybrid storage. Although effective, the vast majority of research is focused on a single file server.

Recently a great of work paid more attention on the data layouts of multiple heterogeneous servers. S4D-Cache [5], [31] uses all SSD-based file servers as a cache and selectively caches performance-critical data on these high performance servers. CARL [32] selects and places file regions with high access costs onto SSD-based file servers at the I/O middleware layer, but the region cannot be placed onto both SSDs and HDDs. PADP [33] and PSA [34], [35] employ stripe size variation to improve the performance of hybrid PFSs. HAS [36], [37] adaptively selects the optimal data layout for heterogeneous parallel file systems with specific access patterns.

The above mentioned techniques are effective in improving the performance of PFSs. However, there is little effort devoted to data layout considering both heterogeneous servers in a hybrid PFS and complex I/O workloads at different part of a file. Recent work method to overcome such challenges in a hybrid PFS. However, it relies on a prior knowledge of access patterns of application. As opposed to this, this study uses a holistic adaptive file stripe optimization method to address all these issues.

3 Heterogeneity-Aware Region-Level Data Layout

3.1 Overview of HARL

The proposed data layout scheme, HARL, aims to optimize the hybrid PFS layout by using varied-size file stripes instead of fixed size. To accommodate both heterogeneous servers and complex I/O workloads, HARL adopts the idea of "divide and conquer" to achieve the optimal data layout. First, it divides a large file into several small regions according to the I/O workloads such that each region has more similar access patterns. Then, HARL determines the appropriate file stripe sizes on heterogeneous servers based on their storage performance for each region.

Fig. 2b illustrates the idea of the heterogeneity-aware region-level data layout scheme. In this example, HARL divides a file into three adjacent regions and assigns different stripe sizes on HServers and SServers for each region. Specially, since SServers have higher I/O performance, SServers are usually allocated with larger stripe sizes than HServers in each region, so that each server can complete their I/O requests near-simultaneously. Compared with the traditional layout (Fig. 2a), HARL is a fine-grained, adaptive data layout scheme, which can significantly alleviate the load imbalance among heterogeneous servers and improve the hybrid PFS performance.

To obtain the optimal layout scheme, one needs to rely on a prior knowledge of the data access patterns. Fortunately, many data-intensive applications have predictable I/O patterns [19], [38], [39]. For example, the BTIO application [40], an I/O kernel responsible for solving block-tridiagonal matrices on a three dimensional array, has this feature. For BTIO, once the size of the array, the number of time steps, the write interval, and the number of processes are given, the I/O behaviors can be accurately predicted before the program executes. Since the program often run multiple times and these patterns do not fluctuate significantly, it provides an opportunity for HARL to achieve the optimal data layout based on its I/O behavior analysis.

Fig. 3 shows the procedure of HARL, which includes three phases. In the *Tracing Phase*, the runtime statistics of data accesses are collected into a trace file during the application's first execution. In the *Analysis Phase*, by analyzing the I/O trace, the large file is divided into different regions according to the application's I/O characteristics, then each region's stripe sizes are determined based on a data access cost model. In the *Placing Phase*, the file is placed on the underlying heterogeneous servers at runtime with the optimal file stripes obtained in the *Analysis Phase*.

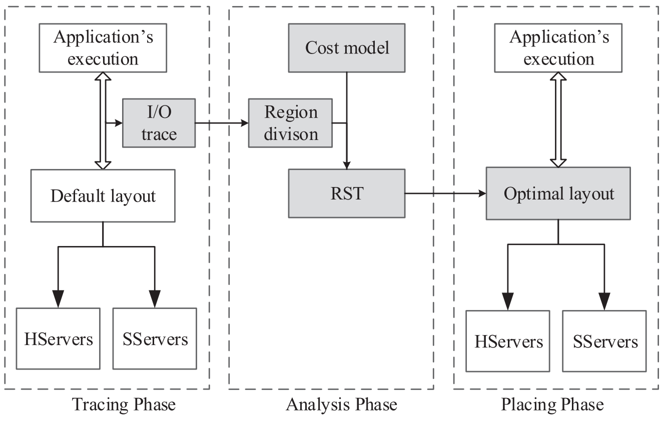


Fig. 3. The procedure for HARL scheme.

Through these three phases, HARL can largely improve the application's I/O performance in subsequent runs.

3.2 I/O Trace Collection

A *trace collector* is responsible for collecting runtime file access information of parallel applications. While there are some techniques and tools that can be used for data analysis, we use IOSIG, which is an I/O pattern collection and analysis tool developed in our previous work [41], to capture the information required by HARL. IOSIG is a pluggable library of MPI-IO, which supports MPI-IO and standard POSIX IO interfaces. IOSIG can help to gather all the information of file operations, including file access type, operation time, and other process related data. After running the applications with the *trace collector*, we can get process ID, MPI rank, file descriptor, type of operation, offset, request size, and time stamp information. To facilitate the region division and guide the optimal data layout, the collector sorts all file read and write requests in ascending order in terms of their offsets.

3.3 File Region Division

Since fixed stripe sizes on servers are unable to provide optimal performance for the whole file, as discussed in Section 1, HARL divides a file into fine-grained regions and applies special stripe size optimization for each region. One may logically divide the address space of a file into regions by a fixed chunk size (e.g., 64 or 128 MB). While this method is simple, it is difficult to select a proper region size that fits diverse I/O patterns in a real system. In contrast, HARL adopts a varied-size region division method, as shown in Algorithm 1.

The algorithm's goal is to identify continuous file chunk accessed with similar I/O patterns, so that a given data layout may benefit more I/O requests. Starting from file offset 0, the algorithm uses *average request size* as a common feature to find the proper delimiting points. It reads the first two entries of the requested size from the trace file and calculates the coefficient of variation (CV), the result of dividing the standard deviation by the average request size in the current sample. It continually adds the next request and calculates the CV until the trace ends. If the new CV value falls close to the previous one, namely, the percentage difference between the new CV value and the previous one is less than 100 percent (line 9), it continues adding the next entry and repeats the calculations. Otherwise, it logs the offset, creates another delimiting point to start a new region, and restarts calculations with a new CV. As a normalized measure of dispersion of data distribution, CV is very sensitive to changes in the average request size and allows us to detect the point where the application changes the I/O behavior. At the end, the algorithm returns a list of file regions with their average request sizes.

Algorithm 1. File Region Division Algorithm

```
Input: Sizes of file requests: $r_0, \ldots, r_{n-1}$; Offset of file
       requests: $o_0, \ldots, o_{n-1}$
Output: Offset of each file region $O_0, \ldots, O_{m-1}$; Average
        request size for each file region: $A_0, \ldots, A_{m-1}$
1  $sum = 0$ ; $cv\_prev = 0$; $reg = 0$ /*region #*/;
2  $reg\_init = 0$ /*The first request served by this region */;
3  for $i = 0; i < n; i++$ do
4      $sum += r_i$;
5      $avg = \frac{sum}{i - reg\_init + 1}$;
6      $std = \sqrt{\sum_{k=reg\_init}^{i} (r_k - avg)^2 / (i - reg\_init + 1)}$;
7      $cv\_new = std/avg$;
8      if $(100 * |cv\_new - cv\_prev|)/cv\_prev < threshold$ then
9          $cv\_prev = cv\_new$;
10     else
11         $sum = 0$ /*Restart with new CV */;
12         $cv\_prev = 0$;
13         /* Set offset and average request size in region: $reg$ */;
14         $O_{reg} = o_{reg\_init}$;
15         $A_{reg} = avg$;
16         $reg\_init = i + 1$ /*The first request served for next
           region will be $i + 1$ */;
17         /*Created region for next region */;
18         $reg++$;
19     end
20 end
```

One potential issue is that this algorithm may generate too many regions, which leads to substantial extra metadata management overhead and compromises the final I/O performance. To overcome this issue, we limit the number of created regions by adjusting the *threshold* value. If the number of the regions is greater than the number from the fixed-size region division, as in the segment-level layout scheme [10], the *threshold* increases from 100 percent to a higher value. This tuning can guarantee that the number of the regions is bounded by the number of the fixed-size region division method [10]. Using a fixed region size of 64 MB as an example, the total regions in a 10 GB file requires at most 160 entries in RST, an acceptable metadata overhead. Furthermore, we combine adjacent regions with the same stripe sizes to reduce the total number of RST entries.

3.4 Access Cost Model

To obtain the optimal stripe size on each server for a given file region, we introduce an analytical model to evaluate the data access time of a file request in a hybrid PFS. The model fully considers the application, the system (architecture, network, and storage), and the layout related characteristics in the data access procedure, and the corresponding parameters are listed in Table 1.

TABLE 1
Parameters in Cost Analysis Model

Application Parameters	
o	Offset of the file request
r	Size of the file request
op	Type of the file request (read or write)
Architecture Parameters	
M	Number of HDD servers (HServers)
N	Number of SSD servers (SServers)
Network Parameters	
t	Unit data network transfer time
Storage Parameters	
α_h^{min}	Minimum startup time on HServer
α_h^{max}	Maximum startup time on HServer
β_h	Unit data transfer time on HServer
α_{sr}^{min}	Minimum startup time for read on SServer
α_{sr}^{max}	Maximum startup time for read on SServer
β_{sr}	Unit data transfer time for read on SServer
α_{sw}^{min}	Minimum startup time for write on SServer
α_{sw}^{max}	Maximum startup time for write on SServer
β_{sw}	Unit data transfer time for write on SServer
Data Layout Parameters	
h	Stripe size on HServer
s	Stripe size on SServer

Note that the storage parameters show distinct features for heterogeneous servers. First, SServer has a much smaller start up time and data transfer time than HServer. This is because SServer does not involve slower mechanical movements. Second, unlike HServer, SServer usually shows different read and write performance because it requires time-consuming garbage collection and wear leveling operations for writes [7]. While simple, this model is sufficient for approximating the general performance profile for heterogeneous servers, as shown in our experiments.

The cost is defined as the I/O completion time of each file request, and it includes three parts. The network transfer time (T_X) is the data transfer time on network, the storage startup time (T_S) refers to the consumption before data operations on storage devices, and the storage transfer time (T_T) is the time spent on actual data read/write operations.

As a file request is usually executed concurrently by multiple sub-requests, the file request cost T is determined by the largest cost of its all sub-requests. Assume the sub-requests are distributed on the m ($m \in [0, M]$) HServers and the n ($n \in [0, N]$) SServers, and the maximal sub-request sizes on HServers and SServers are s_m and s_n respectively, then we can calculate the request cost as follows.

T_X is related with the data size and the network data transfer rate. It is determined by the maximal network transfer cost of all sub-requests on HServers and SServers. Thus

$$T_X = \max\{s_m t, s_n t\}. \tag{1}$$

T_S is determined by the longest startup time on the $m + n$ servers. Let α denote the startup time in each HServer, then the startup time of the m sub-requests can be a variable $X = max(\alpha_1, \alpha_2, \ldots, \alpha_m)$, where α_i ($1 \leqslant i \leqslant m$) has an independent identical distribution as α. Assume α follows an uniform distribution on $[\alpha_h^{min}, \alpha_h^{max}]$, then the probability function of α is $P(\alpha < x) = (x - \alpha_h^{min})/(\alpha_h^{max} - \alpha_h^{min})$, where $x \in [\alpha_h^{min}, \alpha_h^{max}]$, and the probability density function of X is

$$f(x) = \frac{m \times (x - \alpha_h^{min})^{m-1}}{(\alpha_{sr}^{max} - \alpha_{sr}^{min})^m}, \alpha_h^{min} \leqslant x \leqslant \alpha_h^{max}. \tag{2}$$

Hence, the startup time on the m HServers is

$$T_h^S = \int_{\alpha_h^{min}}^{\alpha_h^{max}} x f(x) dx = \alpha_h^{min} + \frac{m}{m+1}(\alpha_h^{max} - \alpha_h^{min}). \tag{3}$$

Similarly, the startup time for read sub-requests on the n SServers is

$$T_{sr}^S = \alpha_{sr}^{min} + \frac{n}{n+1}(\alpha_{sr}^{max} - \alpha_{sr}^{min}). \tag{4}$$

Based on Equations (3) and (4), the overall startup time for a file read request is

$$T_{SR} = \max\{T_h^S, T_{sr}^S\}. \tag{5}$$

T_T is also determined by the maximal storage transfer time of all the sub-requests. For a read request, it can be calculated as

$$T_{TR} = \max\{s_m \beta_h, s_n \beta_{sr}\}. \tag{6}$$

Based on Equations (1), (5), and (6), the overall cost of a file read request is

$$T = T_X + T_{SR} + T_{TR}. \tag{7}$$

The Equations (5), (6) depict the cost for reads, startup and transfer time for writes (T_{SW} and T_{TW}) will be similar except we exchange the read parameters with write. Thus the overall cost of a write request is

$$T = T_X + T_{SW} + T_{TW}. \tag{8}$$

From above equations, we can see that T depends on four parameters: s_m, s_n, m and n, which can be calculated according to the stripe sizes h and s. We assume the file data are distributed from the 0th to the (M+N-1)th server in a round-robin way, and let $S = M * h + N * s$, $r_b = \lfloor o/S \rfloor$, $r_e = \lfloor (o + r)/S \rfloor$, $l_b = o - r_b * S$, and $l_e = (o + r) - r_e * S$, then the server number of the beginning and ending sub-requests are $n_b = (l_b < M * h)?\lfloor l_b/h \rfloor : \lfloor (l_b - M * h)/s \rfloor + M$, $n_e = (l_e < M * h)?\lfloor l_e/h \rfloor : \lfloor (l_e - M * h)/s \rfloor + M$, the size of the beginning and ending fragment are $s_b = (l_b < M * h)?\lfloor h - l_b\%h \rfloor : h - (l_e - M * h)\%s$, $s_e = (l_e < M * h)?\lfloor h - l_e\%h \rfloor : s - (l_e - M * h)\%s$. Based on the locations where the file request begins and ends, the sub-request distributions fall into four cases, as shown in Fig. 4. Due to space limitation, we only describe how to calculate these parameters for case (a) where the request begins and ends at certain HServers. Let $\Delta_r = r_e - r_b$, $\Delta_c = n_e - n_b$, then the four critical parameters are calculated as in Fig. 5. By following the same arguments, we can derive the parameters for other cases.

From the cost model, one can observe that the access time of a file request can be significantly impacted by the server

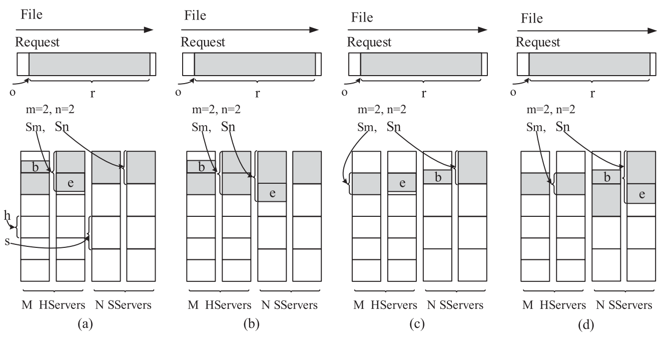


Fig. 4. Four typical cases of file sub-request distribution on servers. (a): Both beginning and ending sub-requests are located on HServers; (b): Beginning sub-request is on HServers but ending sub-request is on SServers; (c): Beginning sub-request is on SServers but ending sub-request is on HServers; (d): Both beginning and ending sub-requests are on SServers.

stripe size h and s, motivating us to optimize them to improve I/O performance.

3.5 Stripe Sizes Determination

Based on the above per-file-request model, HARL uses a heuristic iterative algorithm to find the optimal stripe sizes on HServers and SServers for each region. The goal is to minimize the data access cost of all file requests in that region instead of a single request.

Algorithm 2 shows the procedure of determining the optimal stripe sizes for each region. Starting from h equaling 0, the loop iterates h in 'step' increments while h is less than $\overline{R}$. We use the average request size $\overline{R}$ because we use it to divide the region in Algorithm 1 and it is a good metric to describe the common feature of the workloads. The extreme configuration we do consider is where h is $\overline{R}$, which means dispatching the file request data only on one HServer may obtain better I/O performance. In the second loop, s starts from a size which is larger than h because this configuration can lead to load balance among heterogeneous servers. We also consider the extreme case where the requested data are only placed on one SServer. For each pair of stripe sizes, the loop iterates to calculate the total access cost of all file requests in that region, according to the proposed data access cost in Section 3.4. Note that the request cost is accumulated based on the request type (lines 9 and 12) since the read operations come with different performance as the write operation. Finally, the pair of stripe sizes leading to minimal region access cost (Reg_cost) is chosen. The 'step' value is 4 KB (line 3), which can be chosen by the user. Finer 'step' values result in more precise h and s values, but with increased calculation overhead. However, the computational overhead of this algorithm is acceptable because the calculations are simple arithmetic operations that run off-line.

Condition		Sm	Sn	m	n
Δr=0	Δc=0	s_b	0	Δc +1	0
	Δc=1	max{s_b, s_e}	0	Δc +1	0
	Δc>1	h	0	Δc+1	0
Δr≥1	Δc =0	max{ Δr*h-h+s_b+s_e, Δr*h}	Δr*s	M	N
	n_b+1=M and n_e=0	max{ Δr*h-h+s_b, Δr*h-h+s_e}	Δr*s	Δr =1?2: M	N
	n_b+1!=M or n_e!=0	Δr *h	Δr*s	Δc <-1?(M+1+ Δc):M	N

Fig. 5. The calculation of critical parameters s_m, s_n, m and n in case (a) of Fig. 4. These parameters are related with stripe sizes h and s.

RST

Region #	File_offset	HServer stripe size	SServer stripe size
0	0	16KB	64KB
1	128MB	36KB	144KB
2	192MB	26KB	80KB
⋮	⋮	⋮	⋮

Fig. 6. Data structure of the RST table in HARL scheme.

Algorithm 2. Region Stripe Size Determination

```
Input: File region: $reg$ including file request $R_0, \dots, R_{k-1}$,
       Average request size $\overline{R}$ in $Reg$
Output: optimal stripe sizes: $H$ for HServer, $S$ for SServer
1  $step \leftarrow 4KB$;
2  $opt\_cost \leftarrow \infty$;
3  for $h \leftarrow 0; h \leq \overline{R}; h \leftarrow h + step$ do
4      for $s \leftarrow h + step; s \leq \overline{R}; s \leftarrow s + step$ do
5          for $i \leftarrow 0; i < k; i \leftarrow i + 1$ do
6              $Reg\_cost \leftarrow 0$;
7              if $operation\_type(R_i) = Read$ then
8                  $T_i \leftarrow$ Calculate cost of $R_i$ according to
                   Equation (7);
9              else
10                 $T_i \leftarrow$ Calculate cost of $R_i$ according to
                   Equation (8);
11             end
12             $Reg\_cost \leftarrow Reg\_cost + T_i$;
13         end
14         if $Reg\_cost < opt\_cost$ then
15             $opt\_cost \leftarrow Reg\_cost$;
16             $H \leftarrow h$;
17             $S \leftarrow s$;
18         end
19     end
20 end
```

3.6 Static Region-Level Data Placement

To guide the data placement, the information of optimal stripe sizes for each region is stored into a global *region stripe table* (RST). Fig. 6 shows an example of the RST data structure. In this example, the file consists of multiple regions, and the stripe sizes for the first three regions are (16 KB, 64 KB), (36 KB, 144 KB), and (26 KB, 80 KB) respectively. Although the metadata includes more information, its size is not too large because the number of regions is limited in the region division algorithm in Section 3.3. Moreover, if adjacent regions have the same optimal stripe sizes, the two regions are combined into a larger region. This can further reduce the metadata management overhead.

In the *Placing Phase*, the file is placed on the underlying heterogeneous servers with optimal stripe sizes for each fine-grained file region. A PFS commonly includes three components. The file clients issue requests on behalf the applications, the servers are responsible for storing file data, and the metadata servers (MDS) contain the description information of the files. Upon receiving a file request, a client first contacts MDS to get the file's metadata, then it interacts with servers directly. To perform the optimal data placement, MDSs look up the RST table according to the request's offset and length, and return this information to the client. Then, the client writes the file data on each server with the optimal stripe sizes from RST.

3.7 Implementation

The proposed layout scheme can be implemented either in the PFS or I/O middleware layer. The former solution requires specific file metadata communication between clients and servers to support the region-level striping strategy, which is not currently supported by PFSs. In order to maintain its portability and achieve a simple implementation, HARL is integrated into the I/O middleware layer, which lies above various PFSs.

We implement HARL within MPICH2 [13] that runs on OrangeFS [1]. In the *Analysis Phase*, we use one file server in the parallel file system to test the startup time α and data transfer time β for HServers and SServers with read/write patterns. These parameters can vary with different I/O patterns. In addition, we use a pair of nodes (one client node and one file server) to estimate the network transfer time t. We repeat the tests thousands of times (the number is configurable), and then calculate their average values, which are used as the parameter values.

In the *Placing Phase*, HARL logically maps a large file into multiple OrangeFS files, each representing a separate file region with similar I/O workloads. In MPICH2, a *region-to-file mapping table* (R2F) is used to record the translation from a logical file region to a physical OrangeFS file. For each region, the data is distributed on underlying servers with the optimal stripe sizes stored in RST. This can be implemented by leveraging the existing varied-size striping mechanism in OrangeFS. RST and R2F are stored in the same directory as the applications, and are loaded when `MPI_Init()` is triggered and unloaded when `MPI_Finalize()` is executed. Furthermore, the `MPI_File_read/write()` (and other variants of read/write) are modified, so file requests can be automatically forwarded to the files in the PFSs with optimized stripe sizes.

4 Dynamic Region-Level Data Layout Scheme

In the previous section, we described a static region-level data layout in a hybrid PFS. By allocating optimized stripe sizes for HServers and SServers, this layout scheme is promising to decrease the *overall* data access cost in each region. However, since a static layout is usually favorable for a certain type of access patterns, this static layout still can not reduce the data access cost for *each request* with a various pattern, thus it is not the most efficient way to serve all data accesses. In a practical system, many applications have complex access patterns, it is desirable to develop new data layout schemes to further improve I/O performance.

4.1 Idea of Dynamic Data Layout

To address the issue of the static data layout approach, we propose a dynamic region-level data layout scheme (HARL-D), which leverages data replication to further improve parallel I/O performance at runtime. Since a static layout is only favorable for a limited access patterns, HARL-D creates multiple replicas for each region. For each data access, the strategy dynamically chooses the replica with the lowest access cost to serve the request. Since each request is assigned to the best fit replica, this replication-based dynamic data layout strategy can serve various I/O workloads with higher performance.

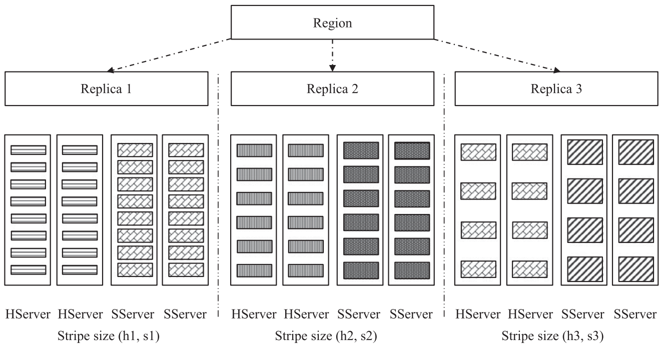

Region
Replica 1
Replica 2
Replica 3
HServer HServer SServer SServer
Stripe size (h1, s1)
HServer HServer SServer SServer
Stripe size (h2, s2)
HServer HServer SServer SServer
Stripe size (h3, s3)

Fig. 7. The dynamic data layout scheme. Each region has three replicas, each with a different stripe size pair (h and s) on HServers and SServers.

For data replication, the first issue is how many replicas (denoted by n) should be created for each region. Although we could create a corresponding replica for each access pattern to achieve perfect performance, it may lead to unacceptable storage cost. For simplicity, we make three replicas for each file region throughout this paper ($n = 3$), as shown in Fig. 7. Of course, one can choose different number of replicas depending on his performance and cost trade-offs.

The second issue is how to determine the optimal layout policy for each replica. Obviously we cannot further decrease the overall I/O cost if we create a replica with a randomly chosen data layout (a pair of h and s). To address this issue, we classify requests in a region into n groups, each with similar access patterns, and then create a replica for each group with an optimal layout based on the representative access characteristics.

Inspired by the data clustering approach in statistics domain [42], we try to find the centers of these groups with an iterative refinement method. The detailed description of the algorithm is as shown in Algorithm 3, where each request is characterized by the request size. As such, all requests can be represented by a set of points in a one-dimensional Euclidean Space. For any point $P_1(x_1)$ and $P_2(x_2)$, their distance can be defined as

$$||P_1 - P_2|| = \sqrt{(x_1 - x_2)^2}. \tag{9}$$

If the number of requests is less than or equal to n, a randomly selected request point is assigned to P_{g_i} as a center of the ith group. Otherwise, each request point is assigned to group G_i whose center is closest to the request point. After all the request points have been processed, the algorithm re-compute the new center for each group. This procedure is repeated until P_{g_i} is no longer changed or three times at most.

Although the computational overhead of the algorithm increases in proportion to the number of requests, the request grouping is an off-line method and it only runs once based on the I/O trace analysis, so the computation overhead in a practical HPC system is acceptable.

The optimal data layout policy (a pair of h and s) for each replica is determined by the access pattern of one center of the group, according to Algorithm 2 in previous Section 3.5. In this case, the input of the algorithm is replaced by the requests in each group and the average request size $\overline{R}$ is replaced by the request size of the center point. Since file requests in each group have closed access patterns, the chosen (h, s) will have high likelihoods to benefit from the given requests for that group. During the subsequent run of the application, the dynamic data layout scheme will estimate the request access cost if it were redirected to the created replicas, and assign it to the corresponding replica with lowest access cost.

Algorithm 3. Iterative Request Grouping

```
Input: Requests: R[$1, i$] in each file region
Output: Group $G_1, G_2, G_3$
1  if ($i \le n$) then
2      for ($\forall i \in [1, n]$) do
3          $P_{g_i} \leftarrow$ randomly selected R[t];
4      end
5  end
6  else
7      $count \leftarrow 0$;
8      while ($P_{g_i}$ is changed$||$count $\le 3$) do
9          $G_i \leftarrow \arg\min_{|G_j|} \{||P_{s_j} - P_{g_i}||\}$;
10         $P_{g_i} \leftarrow \frac{1}{|G_i|} \sum_{P_{s_j} \in G_i} P_{s_j}$;
11         $count++$;
12     end
13 end
```

One concern is how to handle data writes in the subsequent runs of the application. As it involves multiple replicas, we use a lazy synchronization mechanism [38] for data writes to improve performance and keep data consistency. First, we write data to the selected replica. Then, we apply lazy updates to synchronize data from the first replica to other replicas. Hence, we only consider the data access cost on the chosen replica for data writes, and ignore the background data synchronization cost.

4.2 Implementation

The selection of the replica for each file request is based on the cost analysis with the proposed model. We made a prototype of the cost estimation and dynamic data replica selection by modifying several MPI-IO functions.

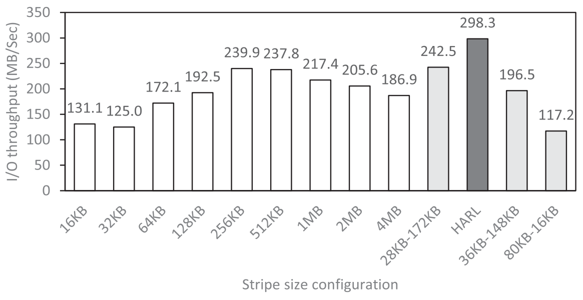
`MPI_File_open`: we open all corresponding replica files, each with a different data layout.

`MPI_File_read`: we first calculate data access costs for all replicas based on the proposed model, and then choose the replica with the lowest cost to handle the request. When data access is finished, the offsets of all replicas are synchronized.

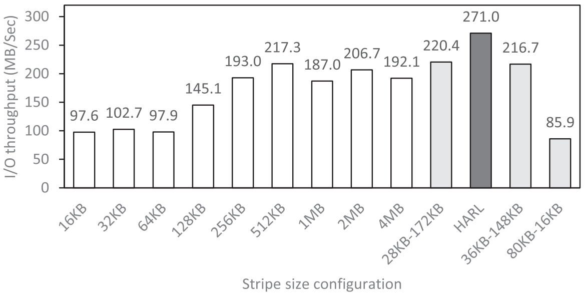
`MPI_File_write`: we handle the request on one replica with the lowest cost, then insert the requests of other replicas into a lazy synchronization queue. To avoid interfering with the normal I/O operations, a dedicated data synchronization thread is implemented to conduct these lazy write requests in the queue. When data access is finished, the offsets of all replicas are synchronized.

`MPI_File_seek`: we calculate the offset and perform seek operations in all opened replica files.

`MPI_File_close`: we synchronize data blocks for all replicas and close all opened replica files.



(a) Read throughput



(b) Write throughput

Fig. 8. Throughputs of IOR with different layouts.

5 Performance Evaluation

5.1 Experimental Setup

The experiments were conducted on a 65-node SUN Fire Linux cluster. Each computing node has two AMD Opteron (tm) processors, 8 GB memory and a 250 GB HDD. The operating system is Ubuntu 13.04 and the parallel file system is OrangeFS v2.8.6. All nodes are equipped with Gigabit Ethernet interconnection, and eight nodes are equipped with additional PCI-E X4 100 GB SSD. Eight nodes are used as computing nodes, eight as HServers, and eight as SServers. All SServers and HServers are accessed through one OrangeFS. By default, six HServers and two SServers are used to build the hybrid OrangeFS file system, and the file is striped over the file servers in a round-robin fashion.

The widely-used parallel file system benchmark IOR [9], BTIO [40], HPIO [43], and an application [11] are used to test the hybrid file system performance.

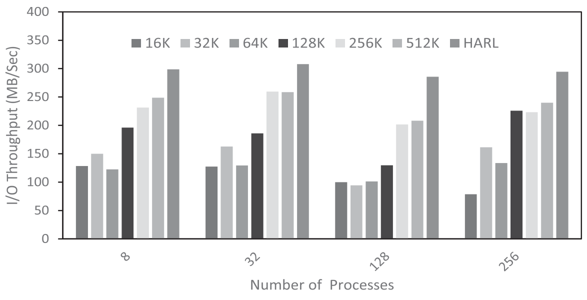
5.2 Evaluation on Static Heterogeneity-Aware Data Layout

In the experiments, we compare three data layout schemes: the fixed-size stripe, the randomly-chosen stripe and the proposed HARL scheme.

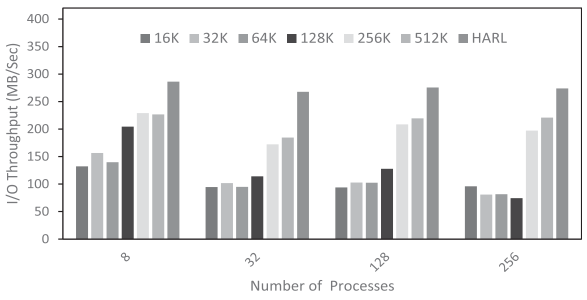
5.2.1 IOR Benchmark

Read and Write Results. The experiments are conducted to compare the I/O performance of the hybrid PFS with the proposed data layout scheme, HARL, and two other strategies, which use a fixed-size or randomly chosen file stripe. For the following tests, IOR benchmark runs with 16 processes, and the request size is kept to 512 KB unless otherwise specified. Each process is responsible for accessing its own 1/16 of a 16 GB shared file and continuously issues requests with random offsets.

Fig. 8 demonstrates the I/O performance of the hybrid file system with different layouts. In this figure, layout '64 KB'

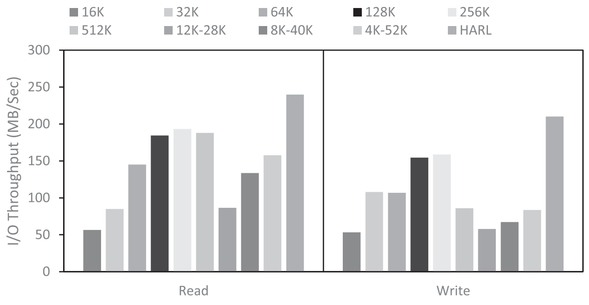


(a) Read throughput

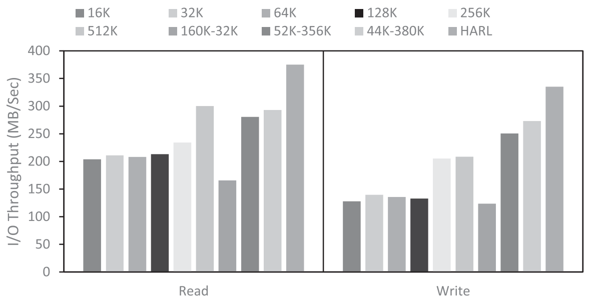


(b) Write throughput

Fig. 9. Throughputs of IOR with various numbers of processes.



(a) 128KB



(b) 1024KB

Fig. 10. Throughputs of IOR with various request sizes.

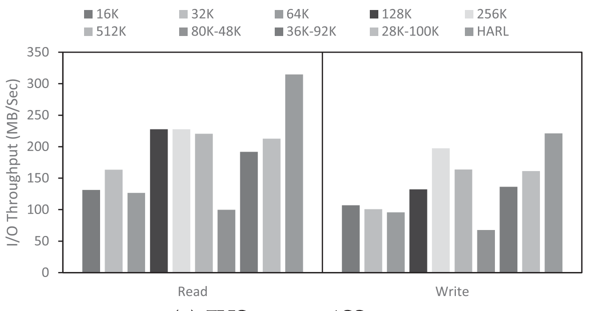
means the stripe sizes are 64 KB for all file servers, and '36 KB-148 KB' means the stripe size is 36 KB for HServers and 148 KB for SServers. The rest of the layouts have similar meaning. It is observed that the proposed heterogeneity-aware layout can achieve I/O performance improvement for both read and write operations. While the performances of the fixed-size and randomly chosen stripe schemes vary with the adopted stripe size, HARL provides the best performance. With the optimal data layout of (32 KB, 160 KB) and (36 KB, 148 KB) for reads and writes respectively, HARL improves the I/O performance by 73.4 and 176.7 percent over the default layout with a fixed-size stripe of 64 KB. Compared with other layouts with different *but* fixed-size stripes, HARL improves the performance up to 138.6 percent for reads and 177.6 percent for writes. Compared with the randomly chosen stripe strategies, the read performance can improve to 154.5 percent and write performance can improve to 215.4 percent. The experiments prove HARL performs optimally and the stripe size determining formula is effective.

Varying Number of Processes. The I/O performance is also evaluated with a varied number of processes. The IOR benchmark is executed with 8, 32, 128 and 256 processes, respectively, at a fixed request size of 512 KB. As shown in Fig. 9, the results are similar to the previous test. HARL improves the I/O performance for both read and write operations. With different number of processes, the I/O throughput increases to 144.1, 141.8, 202.7 and 274.1 percent, respectively, for reads compared with layout schemes with a fixed-size stripe, and 116.4, 182.7, 192.8, and 268.3 percent for writes, respectively. Compared with the default layout (stripe size of 64 KB), the read performance achieves a 144.1, 138.1, 182.3, and 120.2 percent improvements, and write performance achieves a 104.8, 182.2, 168.5, and 235.1 percent improvements. The results illustrate that HARL has high scalability in terms of number of processes.

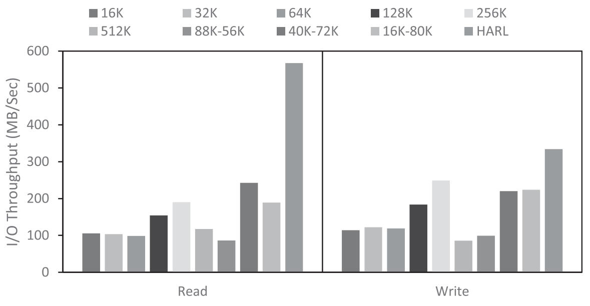
Varying Request Sizes. In Fig. 10, the I/O performance is examined with varied request sizes. The IOR benchmark is executed with request sizes of 128 and 1024 KB. HARL improves read performance from 24.1 to 325.0 percent, and write performance from 32.4 to 293.5 percent, in comparison with conventional layout methods. In terms of the default layout with 64 KB stripe size, HARL achieves an 80.1 percent improvement for read and 147.1 percent for write operations. Compared with layout strategies which use randomly varied stripe sizes, the read performance boosts from 20.6 to 222.3 percent, and write performance increases from 22.7 to 263.1 percent. When the request size is 128 KB, the optimal stripe size pair is (0 KB, 64 KB); thus, distributing the file only on the two SServers offers the highest I/O performance. When the request size is 1,024 KB, HARL distributes the file on both HServers and SServers for higher I/O performance.

Varying Server Configurations. The I/O performance is examined with varied ratios of HServers to SServers. The OrangeFS is built using HServers and SServers with the ratios of 7:1 and 2:6. The request size is kept to 512 KB. Fig. 11 shows the average I/O throughput with different file server configurations. As the results depict, HARL improves I/O performance for both data reads and writes. Read performance increases from 37.6 to 556.1 percent, and write performance improves from 112.2 to 288.7 percent in comparison with other layout methods. Compared with the default layout with a 64 KB stripe size, HARL achieves a 474.9 percent improvement for reads and a 180.3 percent for writes. In the experiments, read and write performance are improved as the number of SServers increased. This is because the I/O performance of SServers is efficiently utilized by the heterogeneity-aware layout scheme. If the number of SServers is small, HARL distributes the file on both SServers and HServers. However, if the number of SServers is greater, the file is placed only on high-performance SServers.

Varying I/O Workloads. All the above results have clearly confirmed the efficiency of HARL with uniform I/O workloads. In the experiments, we evaluated HARL under varied I/O accesses. In order to simulate the complicated



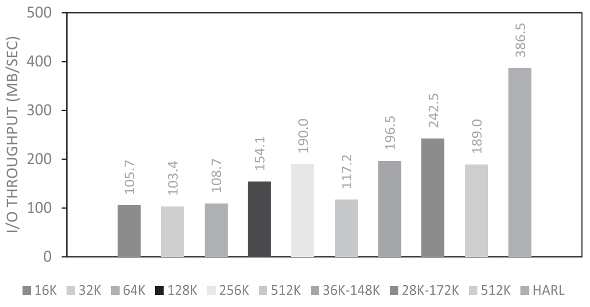
(a) 7HServers : 1SServer



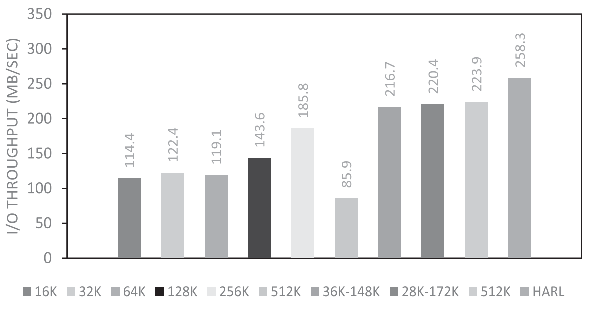
(b) 2HServers : 6SServers

Fig. 11. Throughputs of IOR with various server configurations.

non-uniform I/O workload, we modified IOR benchmark to access a four-region data file. The size of each region is 256, 1,024, 2,048 and 4,096 MB, respectively. For each region, IOR issues requests with different request sizes. Fig. 12 shows the average I/O throughput of the hybrid PFS with different data layout strategies. From the results, it can be easily observed that HARL improves read performance from 59.4 to 265.8 percent, and write performance from 17.2 to 200.7 percent compared with other layout methods. Compared with the default data layout with a 64 KB fixed stripe size, HARL achieves a 255.6 percent improvement for reads and 116.9 percent for writes. The results indicate that the new region-level layout scheme, which divides a file into regions with similar workloads, is capable of increasing performance at a large scale for complex I/O workloads compared with the existing file-level data layout schemes.



(a) Read throughput



(b) Write throughput

Fig. 12. I/O throughputs with non-uniform workloads.

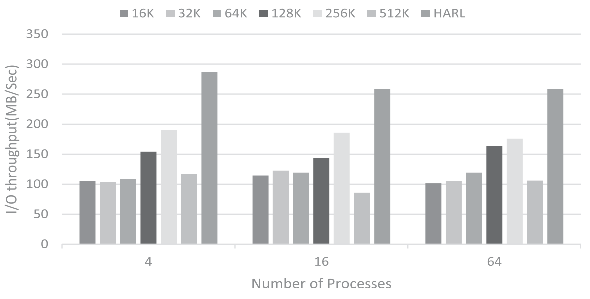


Fig. 13. I/O throughputs of BTIO with different layouts.

5.2.2 BTIO Benchmark

Apart from IOR benchmark above, we also use BTIO benchmark to evaluate HARL. BTIO represents a typical scientific application with interleaved intensive computation and read/write mixed I/O phases. BTIO uses a Block-Tridiagonal (BT) partitioning pattern to solve the three-dimensional compressible Navier-Stokes equations. We consider the Class A and full subtype BTIO workload in the experiments. That is, BTIO writes and reads a total size of 1.69 GB data with collective I/O functions. We use 4, 16, and 64 compute processes since BTIO requires a square number of processes. Output file is striped across six HServers and two SServers. Fig. 13 displays the aggregated I/O throughputs. Compared with the default layout with 64 KB stripe size, HARL achieves 163.5, 116.9, and 114.8 percent improvement with 4, 16, 64 processes, respectively. For other varied but fixed-size striping methods, HARL also demonstrates performance advantages.

5.2.3 Real Application

Finally, the proposed layout is evaluated with a real application's I/O trace, called 'Anonymous LANL App 2' [11]. In this application, each process issues I/O requests in a non-uniform way at different parts of a shared file. In the first part of the file, the request size of each process is relatively small and barely varies. In the following part of the file, request sizes are very small. In the last part, each process issues requests of 131,072 bytes and 131,056 bytes iteratively. The data accesses of this application were replayed according to the I/O trace to simulate the same data access scenario. In the experiment, eight nodes are clients, six nodes are HServers and two nodes are SServers. For this application, HARL recognizes three different regions where the application's I/O behavior is similar. From Fig. 14 we can conclude that HARL can achieve 30.3 to 91.1 percent performance improvement compared to data layouts with a fixed-size stripe. The results indicate that the proposed adaptive data layout is an effective performance optimization strategy for applications with non-uniform I/O workloads.

5.3 Evaluation on Dynamic Region-Level Data Layout

We compare HARL-D with HARL to verify the need of dynamic data layout when applications have various patterns in different parts of a file.

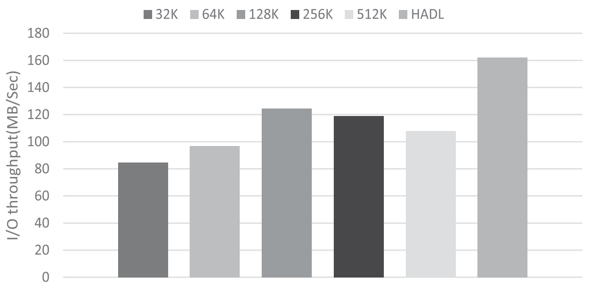


Fig. 14. Performance of LANL App2 with different layouts.

5.3.1 The IOR Benchmark

We modify IOR benchmark to access a three-region data file. Within each region, IOR issues requests with size of 8, 64 and 512 KB. Fig. 15 shows the average I/O throughput of the hybrid PFS with different data layout strategies when we vary the process number from 8 to 256. From the results, it can be easily observed that HARL-D improves read performance from 32.4 to 37.9 percent, and write performance from 27.6 to 36.2 percent compared with HARL. This is because HARL only provides one replicas for requests in each region, which can not bring the best performance for all requests since they have different request sizes. However, HARL-D provides three replicas and it redirects requests to the proper replicas with lowest access costs, thus it is capable of further increasing performance of parallel PFSs.

5.3.2 The HPIO Benchmark

We also modify HPIO to simulate the complex access patterns. HPIO can generate various data access patterns by changing three parameters: region count, region spacing, and region size. We set the region count to 2048 and the region spacing to 0. We vary the region size with 256 and 512 KB. Fig. 16 shows I/O throughputs of HPIO in terms of 16, 32, 64, and 128 processes. Similar to the IOR tests, HARL-D shows better performance than HARL, but the improvement is not as substantial as that of IOR. This is because the variation of access pattern of HPIO is not as significant as that of IOR, which can benefit more from multiple replicas. However, HARL-D still exhibit moderate performance improvement over HARL.

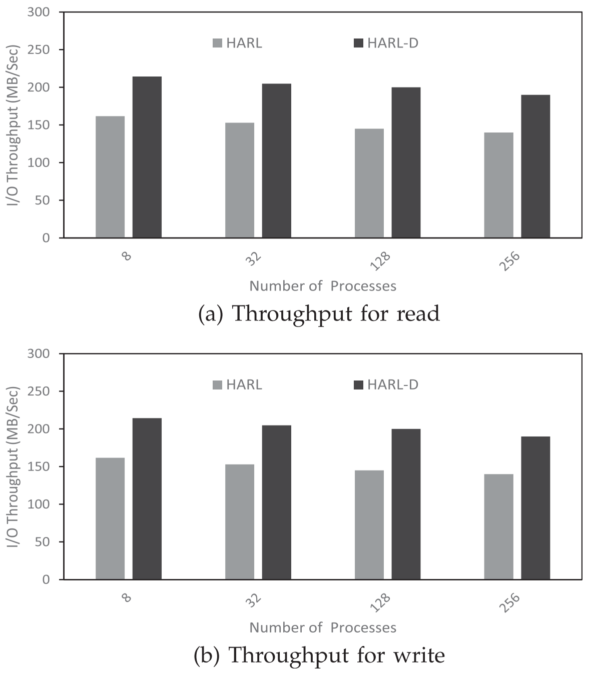


Fig. 15. I/O throughputs with varying numbers of processes.

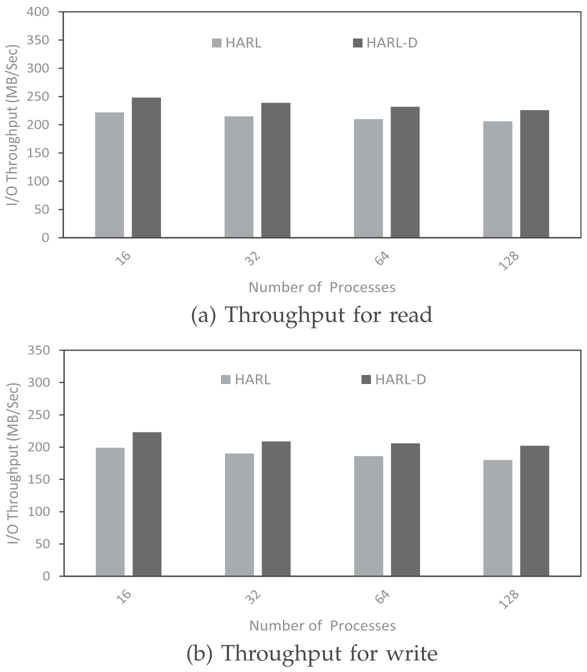


Fig. 16. Throughputs of HPIO with various numbers of processes.

5.3.3 Real Application

We evaluate HARL-D with the real application 'Anonymous LANL App 2'. Fig. 17 shows the performance results. Similar to previous tests, we find that HARL-D outperforms HARL: it obtains 12.8 percent performance improvement compared to HARL. This indicates for some data-intensive real applications, the dynamic data layout scheme exhibits performance advantages over the static data layout scheme.

5.4 Discussion

While making all file servers to complete their I/O accesses near-simultaneously, HARL would potentially lead to more storage space consumption on SServers. Fortunately, most file systems fail to fully utilize the storage space in the underlying devices [44]. In a practical system, this issue is not frequently encountered since the capacities of current SSDs are increasing quickly. In the worst case, where there is a possibility of an SServer running out of space, we could use a data migration method to balance the storage space by

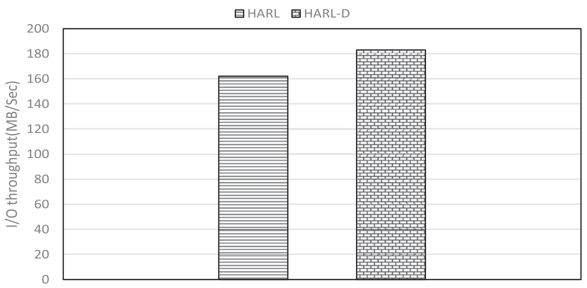


Fig. 17. Performance of LANL App2 under HARL-D.

moving data from SServers to HServers, so the remaining available space on SServers can be guaranteed for new incoming requests. This problem can also be addressed by selectively storing users' performance-critical data in a hybrid PFS, while storing the remaining data in a traditional PFS on HServers.

Although HARL is currently implemented for a single application, it can be also applied to multiple applications with varied I/O workloads. We identify the I/O access patterns at the MPI file level, and do not distinguish between requests coming from the same application or from different applications. For the latter case, we may apply our method to different workloads separately to find their individual data access patterns.

6 Conclusion

We propose a (HARL) data layout scheme, which distributes data across HDD and SSD servers considering application workload and server I/O performance. HARL divides a file into fine-grained regions according to I/O workloads, and adopts varied stripe sizes on HServers and SServers for each region based on the server performance. Furthermore, we develop a dynamic data layout scheme (HARL-D), which creates multiple replicas for each region and redirects file requests to the proper replicas to reduce the access cost at the runtime. In essence, HARL provides an improved matching between the data access characteristic of applications and the data handling capability of file servers in a hybrid PFS. Experimental results show that HARL significantly improves the performance of hybrid PFSs over the fixed-size and randomly-chosen striping methods: the I/O performance improves from 20.6 to 556.1 percent for reads and 22.7 to 288.7 percent for writes, and demonstrates the advantages of HARL-D over HARL.

In the future, we will extend our cost model to accommodate more than two server performance profiles. Another direction is to explore on-line data layout and data migration methods to make heterogeneous I/O systems more intelligent and efficient. Furthermore, we plan to exploit the potential of our approach in distributed systems, e.g., Hadoop and Spark.

Acknowledgments

This work is supported in part by the National Science Foundation of China under Grant No.61572377, the Natural Science Foundation of Hubei Province of China under Grant No. 2014CFB239, the Open-fund from HPCL under Grant No. 201512-02, and the Open Fund from SKLSE under Grant No. 2015-A-06. Chengzhong Xu is the corresponding author.

References

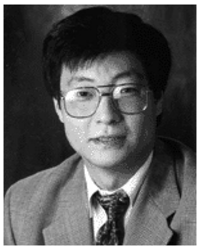
[1] Orange file system. (2016). [Online]. Available: http://www.orangefs.org/
[2] Sun Microsystems, "Lustre file system: High-performance storage architecture and scalable cluster file system," Tech. Rep. Lustre File System White Paper, 2007.
[3] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, pp. 231–244.
[4] D. Nagle, D. Serenyi, and D. Serenyi, "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proc. ACM/IEEE Conf. Supercomputing*, 2004, Art. no. 53.
[5] 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.
[6] 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 Performance Distrib. Comput.*, 2011, pp. 37–48.
[7] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. 11th Int. Joint Conf. Meas. Model. Comput. Syst.*, 2009, pp. 181–192.
[8] M. Zhu, G. Li, L. Ruan, K. Xie, and L. Xiao, "HySF: A striped file assignment strategy for parallel file system with hybrid storage," in *Proc. IEEE Int. Conf. Embedded Ubiquitous Comput.*, 2013, pp. 511–517.
[9] Interleaved Or Random (IOR) Benchmarks. (2016). [Online]. Available: http://sourceforge.net/projects/ior-sio/
[10] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "A segment-level adaptive data layout scheme for improved load balance in parallel file systems," in *Proc. 11th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2011, pp. 414–423.
[11] Application I/O Traces: Anonymous LANL App2, 2014. [Online]. Available: http://institutes.lanl.gov/plfs/maps/
[12] 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.
[13] A. N. Lab, "MPICH2: A high performance and widely portable implementation of MPI," (2015). [Online]. Available: http://www.mcs.anl.gov/research/project-detail.php?id=2y
[14] 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.
[15] A. Ching, A. Choudhary, K. Coloma, L. Wei-keng, R. Ross, and W. Gropp, "Noncontiguous I/O accesses through MPI-IO," in *Proc. 3rd IEEE/ACM Int. Symp. Cluster Comput. Grid*, 2003, pp. 104–111.
[16] A. Ching, A. Choudhary, W.-K. Liao, R. Ross, and W. Gropp, "Efficient structured data access in parallel file systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2003, pp. 326–335.
[17] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Sci. Program.*, vol. 5, no. 4, pp. 301–317, 1996.
[18] J. Bent, et al., "PLFS: A checkpoint filesystem for parallel applications," in *Proc. Conf. High Performance Comput. Netw. Storage Anal.*, 2009, pp. 1–12.
[19] Y. Wang and D. Kaeli, "Profile-guided I/O partitioning," in *Proc. 17th Annu. Int. Conf. Supercomputing*, 2003, pp. 252–260.
[20] S. Rubin, R. Bodik, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 140–153, 2002.
[21] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost effective storage using extent based dynamic tiering," in *Proc. 9th Conf. File Storage Technol.*, 2011, pp. 273–286.
[22] 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.
[23] H. Song, H. Jin, J. He, X.-H. Sun, and R. Thakur, "A server-level adaptive data layout strategy for parallel file systems," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2012, pp. 2095–2103.
[24] Z. Gong, et al., "PARLO: PArallel run-time layout optimization for scientific data explorations with heterogeneous access patterns," in *Proc. 13th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2013, pp. 343–351.
[25] T. Cortes and J. Labarta, "Taking advantage of heterogeneity in disk arrays," *J. Parallel Distrib. Comput.*, vol. 63, no. 4, pp. 448–464, 2003.
[26] T. Pritchett and M. Thottethodi, "SieveStore: A highly-selective, ensemble-level disk cache for cost-performance," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 163–174.
[27] X. Zhang, K. Liu, K. Davis, and S. Jiang, "iBridge: Improving unaligned parallel file access with solid-state drives," in *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 381–392.
[28] Q. Yang and J. Ren, "I-CASH: Intelligently coupled array of SSD and HDD," in *Proc. IEEE 17th Int. Symp. High Performance Comput. Archit.*, 2011, pp. 278–289.

[29] F. Chen, D. A. Koufaty, and X. Zhang, “Hystor: Making the best use of solid state drives in high performance storage systems,” in *Proc. Int. Conf. Supercomputing*, 2011, pp. 22–32.
[30] X. Wu and A. N. Reddy, “Exploiting concurrency to improve latency and throughput in a hybrid storage system,” in *Proc. IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2010, pp. 14–23.
[31] S. He, X.-H. Sun, and Y. Wang, “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.
[32] 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.
[33] S. He, X.-H. Sun, B. Feng, and F. Kun, “Performance-aware data placement in hybrid parallel file systems,” in *Proc. 14th Int. Conf. Algorithms Archit. Parallel Process.*, 2014, pp. 563–576.
[34] S. He, Y. Liu, and X.-H. Sun, “A performance and space-aware data layout scheme for hybrid parallel file systems,” in *Proc. Data Intensive Scalable Comput. Syst. Workshop*, 2014, pp. 41–48.
[35] S. He, Y. Liu, Y. Wang, X.-H. Sun, and C. Huang, “Enhancing hybrid parallel file system through performance and space-aware data layout,” *Int. J. High Performance Comput. Appl.*, vol. 30, no. 4, pp. 396–410, 2016.
[36] 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.
[37] S. He, Y. Wang, and X.-H. Sun, “Boosting parallel file system performance via heterogeneity-aware selective data layout,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2492–2505, Sep. 2016.
[38] 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.
[39] 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.
[40] The NAS parallel benchmarks, 2014. [Online]. Available: www.nas.nasa.gov/publications/npb.html
[41] 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.
[42] J. A. Hartigan and M. A. Wong, “Algorithm as 136: A k-means clustering algorithm,” *J. Roy. Statistical Soc.*, vol. 28, no. 1, pp. 100–108, 1979.
[43] 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. 69.
[44] P. Steege, “50% storage utilization: Are data centers half empty or half full?” 2014. [Online]. Available: http://storageeffect.media.seagate.com/2009/01/storage-effect/50-storage-utilization-are-datacenters-half-empty-or-half-full/

Shuibing He received the PhD degree in computer science and technology from Huazhong University of Science and Technology, China, in 2009. He is now an associate professor in the computer school of Wuhan University, China. His current research areas include parallel I/O systems, file and storage systems, high-performance computing and distributed computing. He has more than 30 papers to his credit in major journals and international conferences including the *IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers*, ICDCS, IPDPS, ICPP, CLUSTER, HiPC and ICA3PP.

Yang Wang received the BSc degree in applied mathematics from Ocean University of China, in 1989, and the MSc and PhD degrees in computer science from Carleton University, in 2001 and University of Alberta, Canada, in 2008, respectively. He is currently in Shenzhen Institute of Advanced Technology, Chinese Academy of Science, as a professor. His research interests include cloud computing, big data analytics, and java virtual machine on multi-cores.

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. He is a distinguished professor in the Department of Computer Science, Illinois Institute of Technology (IIT), Chicago. He is the director of the Scalable Computing Software laboratory, IIT, and is a guest faculty in the Mathematics and Computer Science Division, Argonne National Laboratory. His research interests include parallel and distributed processing, memory and I/O systems, software systems, and performance evaluation and optimization. He is a fellow of the IEEE.



Chengzhong Xu received the PhD degree from the University of Hong Kong, in 1993. He is currently the director of the Institute of Advanced Computing and Data Engineering, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interest includes parallel and distributed systems and cloud computing. He received the Faculty Research Award, Career Development Chair Award, and the Presidents Award for Excellence in Teaching of Wayne State University. He also received the Outstanding Oversea Scholar award of NSFC. He is a fellow of the IEEE.