A Segment-Level Adaptive Data Layout Scheme for Improved Load Balance in Parallel File Systems

Huaiming Song, Yanlong Yin, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
{huaiming.song, yyin2, sun}@iit.edu

Rajeev Thakur, Samuel Lang
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{thakur, slang}@mcs.anl.gov

***Abstract*—Parallel file systems are designed to mask the ever-increasing gap between CPU and disk speeds via parallel I/O processing. While they have become an indispensable component of modern high-end computing systems, their inadequate performance is a critical issue facing the HPC community today. Conventionally, a parallel file system stripes a file across multiple file servers with a fixed stripe size. The stripe size is a vital performance parameter, but the optimal value for it is often application dependent. How to determine the optimal stripe size is a difficult research problem. Based on the observation that many applications have different data-access clusters in one file, with each cluster having a distinguished data access pattern, we propose in this paper a segmented data layout scheme for parallel file systems. The basic idea behind the segmented approach is to divide a file logically into segments such that an optimal stripe size can be identified for each segment. A five-step method is introduced to conduct the segmentation, to identify the appropriate stripe size for each segment, and to carry out the segmented data layout scheme automatically. Experimental results show that the proposed layout scheme is feasible and effective, and it improves performance up to 163% for writing and 132% for reading on the widely used IOR and IOzone benchmarks.**

***Keywords*-parallel file system, adaptive data layout, file segment, optimal stripe size, I/O performance**

I. Introduction

Data-intensive applications widely exist in the domains of scientific computing and engineering simulation, such as nanotechnology, astrophysics, climate, and high-energy physics. The data size and storage volume of these applications continue to grow rapidly. A single disk has reached several terabytes and an individual system has reached petascale or even beyond. However, the improvement rate of disk bandwidth has not kept pace with the growth of capacity. Even worse, I/O performance lags far behind computing capacity, resulting in processors having to wait a large number of cycles for data arrival. Data access performance is often the bottleneck for data-intensive and high-performance computing systems.

Parallel file systems, such as Lustre[1], GPFS[2], pNFS[3], PVFS2[4], and PanFS[5][6], provide a high degree of I/O parallelism for parallel applications by combining a large number of storage devices and utilizing them in concert. In a parallel file system, a large data file is usually striped across multiple I/O servers by using a fixed stripe size in a round-robin manner. This striping method has the benefit of concurrent data access to/from multiple I/O servers and can provide balanced data size in these storage nodes. The data access pattern and the striping method have an interrelated effect on I/O bandwidth in parallel file systems [7][8][9]. One stripping manner can benefit applications with a few specific data access patterns, and different data access patterns require different stripe sizes for higher I/O performance.

Generally, the optimal stripe size for an application depends on its data access patterns. However, for many data-intensive applications, data accesses are non-uniform in a large file. Their request size can be large when accessing one part of the file and small at another part; some parts are accessed repeatedly while others may seldom be accessed. Also, the number of concurrent processes can change with time. Due to the non-uniform data access, one stripe size for a whole file cannot serve all I/O requests with high performance. One stripe size may be suitable for some data access patterns, but not suitable for others. Therefore, it is not always easy to find an optimal stripe size for data-intensive applications, especially for those with variable data access patterns.

Historically, parallel I/O applications and parallel file systems are designed and developed separately. This separation enhances the transparency between the two layers and eases the software implementation. Nevertheless, because of lacking information of each other, data access from client side may mismatch with physical data layout on I/O servers. This mismatch can introduce I/O workload imbalance problem among multiple I/O servers, which is a critical restrict of performance for parallel I/O systems. Figure 1 shows an example of I/O workload distribution of stride I/O, which is a most common occurrence in parallel I/O workloads [10]. The request size is 4 KB and stride size is 12 KB. There are 4 I/O servers. From the figure we can see that, if the stripe size is 4 KB or 8 KB, the data accesses fall only into 1 or 2 I/O servers. However, the data access workload can be evenly distributed on all I/O servers if the stripe size is set to 1 KB or 16 KB. This example demonstrates a potential optimization opportunity by tuning stripe size to balance I/O workload, which can benefit the overall performance of parallel I/O systems.

In this paper we propose a segment-level data layout scheme to optimize the I/O performance for applications with non-

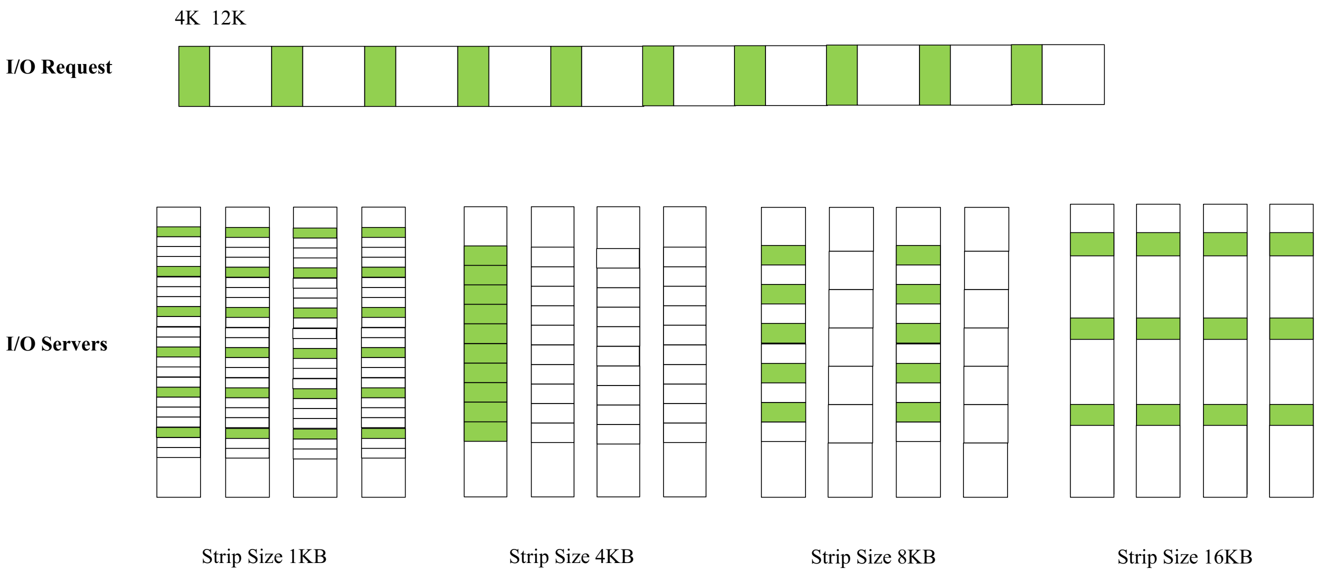


Fig. 1. Stride I/O and data access workload distribution under different layout manners

uniform data access patterns, as well as to address I/O workload imbalance problems. Specifically, we make the following contributions.

- We propose a segment-level layout scheme, which divides files into segments and adopts different stripe sizes for different file segments, in order to improve I/O bandwidth in each segment.
- We describe how to calculate the optimal stripe size for each file segment based on analysis of data access cost.
- We further adjust the stripe size for each file segment to improve I/O workload balance.

The rest of this paper is organized as follows. Section II introduces the related work of data access and layout optimization for parallel I/O systems. In Section III, we propose a segment-level adaptive data layout scheme, and describe a five-step approach to conduct the segmentation. Section IV introduces how to calculate the optimal stripe size for each file segment, and Section V presents how the new approach addresses I/O workload imbalance issues. Experimental results are presented and analyzed in Section VI. Finally, Section VII concludes the paper.

II. Related Work

Data access patterns and layout strategies are two key factors that affect I/O performance in parallel I/O systems. In the last few years, numerous techniques have been proposed in both data access and layout optimization to improve parallel I/O performance. These techniques have been pursued in attacking the I/O bottleneck problem in the form of runtime I/O libraries, middleware, and parallel file systems.

A. Data Access Optimization

Research efforts on data access optimization have focused on reordering of requests. A large body of research has been done in parallel I/O libraries, such as data sieving [11][12], two-phase I/O [13][14], collective I/O [12][15][16], list I/O [17], and datatype I/O [18]. These techniques mainly focus on collecting and merging small or non-contiguous I/O requests into large or contiguous ones, and have achieved I/O performance improvement by reducing the total amount of network packing/unpacking, or disk head thrashing. Besides these request re-arrangement approaches, other research efforts have been devoted to caching[19][20][21] and prefetching[22][23][24][25] at different layers, to take advantage of data fetching beforehand or re-access afterwards.

In order to explore sustained peak I/O performance on storage nodes, a collection of I/O scheduling techniques have been developed on the server side of parallel file systems, including disk-directed I/O [26], server-directed I/O [27], and stream-based I/O [28][29]. Disk-directed I/O [26] sorts the physical blocks into some optimal access ordering and uses double-buffering to overlap disk read/write and network transmission. Server-directed I/O [26] is a derivative of disk-directed I/O. It utilizes a high-level multidimensional data set interface and applies disk-directed techniques at the logical or file level. Stream-based I/O attempts to address the problem of network bottleneck in parallel I/O systems. All these techniques succeed in achieving high bandwidth in disks and networks of I/O servers, either by reducing the frequency of disk seeks, or by reducing the waiting time of network connections.

B. Data Layout Optimization

Most parallel file systems, such as Lustre[1], GPFS[2], pNFS[3], PVFS2[4], and PanFS[5], provide several data layout policies for different I/O workloads. Therefore, parallel I/O performance can also be optimized by reorganizing data layout on I/O servers to create a storage layout consistent with expected data access patterns. A collection of research efforts have been devoted to physical data layout optimization

[30][31][32][33][34][35][36] among multiple I/O servers. Data partitioning[30][31] and replication[34][35][36] techniques are commonly utilized to improve data locality on I/O servers or to reduce the number of disk head seeks. For example, Zhang et al[36] proposed a data replication scheme to amortize I/O workload to multiple replicas to improve the performance, so that each I/O node only serves requests from one or a limited number of processes.

In order to explore parallel I/O more efficiently, data layout optimization has to rely on the prior knowledge of data access patterns. Workload studies on a number of platforms have shown that I/O workloads for most scientific applications usually fall into several patterns [10]. Moreover, there have been some I/O trace techniques and tools [37][25][38] that can be used for data access analysis. Our previous work [7][9] proposed a model to estimate data access costs for different layout manners based on data access patterns, thus to optimize data layout on I/O servers.

Both requests rearrangement and data reorganization techniques have been successful in improving I/O performance for parallel I/O systems. However, to the best of our knowledge, little effort has been done on segment-level data layout optimization. In addition, the proposed layout scheme also takes data access workload balance into consideration, which is another significant difference from others' work.

III. Adaptive Layout Optimization

A. Segment-level Adaptive Data Layout

Data access patterns and layout manners have an interrelated effect on I/O bandwidth in parallel file systems. Usually, data layout optimization on storage nodes must consider the data access patterns from the client side. However, the data access pattern may vary at different parts of a large file. The request size might be very large for some data and very small for others. As a result, one fixed stripe size for the whole file cannot serve all data accesses with high efficiency. We propose a segment-level layout scheme, that divides a large file into several small segments and adopts different layout manners for different segments according to the data access patterns on them, to explore potential benefit for all I/O requests.

In general, for the segments with many small I/O requests, it is better to set a relatively small stripe size, so that one request can be directed to one or a few storage nodes, and different requests can be directed to different I/O servers to avoid extreme access to one or a few nodes. For the segments with large request sizes, it is better to stripe data with a moderately large stripe size, so that applications can benefit from large contiguous data access on I/O servers. Moreover, the number of data blocks involved in one request should not be too large, which can simplify the data collecting/scattering at the parallel I/O middleware or client side library. The proposed data layout scheme chooses optimal stripe size for each file segment, which can improve the aggregate I/O bandwidth for both small and large I/O requests.

Many data-intensive and high-performance computing systems have regular access patterns while accessing data files.

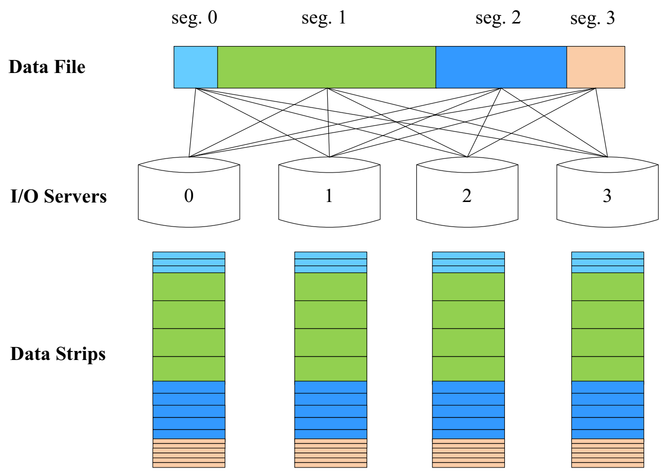


Fig. 2. An instance of the segment-level adaptive data layout scheme. In this example, the file is divided into 4 segments with their different stripe sizes.

For these applications, data access patterns are similar to previous runs. It provides an opportunity to optimize data layout based on the I/O trace analysis. The basic approach of the proposed layout scheme includes the following five steps.

(1) Logically divide the address space of the file into segments by a fixed chunk size (e.g. 64MB or 128MB) for further analysis.
(2) Calculate the average request size of all I/O requests located on each segment according to the I/O traces. If the start offset of an I/O request falls into the segment, the request is counted toward the average request size.
(3) Analyze data access cost for each segment according to our parallel I/O cost model (described in Section IV), and calculate the optimal stripe size for each segment.
(4) Measure data access workload on all I/O servers for each segment, and then choose a proper stripe size close to the optimal stripe size determined in Step (3) to balance workload among I/O servers.
(5) Combine adjacent segments into a larger segment if the two have the same optimal stripe size.

The details of how to calculate the optimal stripe size for each file segment in Step (3) and how to balance the data access workload in Step (4) are described in the following sections. After the above five steps, an optimized segment-level layout can be determined for a large file, with optimal stripe sizes for its individual segments. Figure 2 shows an example of the segment-level data layout scheme. In this example, the file is divided into four segments, and different segments use different stripe sizes. Since the stripe size for each segment is optimized according to the data access pattern in it, the proposed layout scheme can serve all I/O requests with high performance. Compared with existing layout strategies that adopt one stripe size for the whole file, the segment-level layout scheme is a fine-grained optimization, and it is more suitable for applications with complex data access patterns.

The five-step approach is based on I/O trace analysis to conduct the file segmentation. Therefore, the proposed data layout scheme makes a better integration of data access pattern on the client side and data organization on the server side. It needs prior knowledge of data access patterns of applications to make a segment-level optimization. Trace-based data access analysis is widely used for I/O optimization [25][38][39], and such traces can be easily collected at either the application level or I/O library level [25][38]. In our previous work, we have developed a data access trace collector in the MPI-IO library, which can record I/O requests for all MPI processes when a parallel application runs [25].

B. Layout Metadata

By adopting the proposed segment-level layout scheme, a large file is divided into several segments each with a potentially different stripe size. Therefore, the metadata of data layout should include the information about all segments and stripe sizes. We design an <offset, strip_size> pair list to describe the layout information. For example, a list <0, 4KB: 64MB, 64KB: 128MB, 1MB: 1024MB, 64KB> means that the file consists of 4 segments, and stripe sizes for them are 4 KB, 64 KB, 1 MB, and 64 KB, respectively.

Although the metadata includes more information on data layout, the size of metadata is not too large. For example, Google file system (GFS)[40] and Hadoop distributed file system (HDFS)[41] use 64 MB chunk size, and one metadata server can hold all block information of a petascale distributed file system. The proposed layout scheme adopts 64 MB or 128 MB in Step (1) to divide a large file, hence the total number of segments for the whole file system is not too large. Moreover, in Step (5), adjacent segments with same stripe size will be combined together, which can significantly reduce the total number of segments for a file.

The proposed data layout scheme adopts different layout manners for different file segments. The five-step approach is simply a calculation carried out offline to figure out the file segments and layout manner for them. Also, parallel file systems could provide a smart interface to apply the segment-level data layout. A feasible way is to provide layout templates for specific applications. Layout templates could be generated by I/O trace file analysis, and users can choose proper layout templates when a new file is created. For some well-known applications with typical data access patterns, parallel file systems could also provide some specific templates without trace analysis. The proposed scheme provides a more fine-grained approach to enrich and optimize the data layout of files in parallel file systems. Besides, if a file is not specified with any layout templates, it would inherit the layout manner of the directory, or use the default manner.

IV. How to Calculate the Optimal Stripe Size

The proposed data layout scheme uses an optimal stripe size for each file segment, and the optimal stripe size is calculated based on data access cost analysis. In this section, we propose a model to calculate the data access completion time according to the stripe sizes. We can use this model to calculate the optimal stripe size with the lowest completion time for each segment. For simplicity, assume the completion time of each data access on I/O servers include two parts: startup time and read/write time.

$$T_{IO} = T_{startup} + T_{rw} \tag{1}$$

Here the startup time means disk seek and software overhead of the I/O servers, and it is independent from the request size. Read/write time means the time consumption of data read/write from/to disk devices, and this part is proportional to the data size. For each data access, assume the startup time on each I/O server is α. Usually, startup time is not a constant in a storage node with block devices, and we assume it follows uniform distribution between T_{min} and T_{max}, where T_{min} and T_{max} represent the minimum and maximum startup time, respectively. Thus α is the expectation of startup time, and $\alpha = \frac{1}{2}(T_{min} + T_{max})$. If I/O clients access data on k I/O servers in parallel, the overall startup time should be the maximum of all the k I/O servers. Hence, the expectation of the maximum startup time on the k I/O servers can be expressed as follows.

$$T_{startup} = T_{min} + \frac{k}{k+1}(T_{max} - T_{min}) \tag{2}$$

Assume the transmission time of a single unit of data is β, stripe size is s, the request size is r, and the number of I/O servers is n. Therefore, data access time of an I/O request can be expressed as the following three cases.

$$\begin{cases} \alpha + r\beta & (s > r) \\ T_{min} + \frac{\lceil \frac{r}{s} \rceil}{\lceil \frac{r}{s} \rceil + 1}(T_{max} - T_{min}) + s\beta & (\frac{r}{n} \le s \le r) \\ T_{min} + \frac{n}{n+1}(T_{max} - T_{min}) + \frac{r}{n}\beta & (s < \frac{r}{n}) \end{cases} \tag{3}$$

In order to get the minimal data access cost, we compare the minimal completion time of the three cases. In a given system, the completion time of the first and third cases are fixed if the request size r is determinate. For the second case, the minimal value can be achieved when the derivative of s is 0 or at the boundary conditions. The derivative of s can be roughly expressed as follows.

$$\beta - \frac{r(T_{max} - T_{min})}{(r+s)^2}$$

Let the derivative equal 0, then we can calculate the value of s.

$$s = \sqrt{\frac{r(T_{max} - T_{min})}{\beta}} - r \tag{4}$$

If the value of s in Formula 4 also satisfies the precondition $\frac{r}{n} \le s \le r$, minimal completion time can be calculated at that point. Otherwise the minimal completion time can be achieved where s is $\frac{r}{n}$ or r, the end points of boundary conditions. Therefore, we can determine the optimal stripe size s where Formula 3 gets the minimal value.

In Formula 4, we use average request size on each segment to calculate the optimal stripe size. The average request size may have lost some individual information of data access, but it is more representative in most cases. Besides, it can substantially simplify the calculation.

V. How to Balance I/O Workload

The striping manner can easily distribute a file evenly on multiple I/O servers. However, due to non-uniform data access, especially large amounts of non-contiguous data access, balanced data size in all I/O servers does not mean balanced I/O workload on them. The proposed segment-level data layout scheme also takes I/O workload balance into consideration. In this section, we discuss how to achieve this balance by tuning the stripe size.

A. Data Access Workload

In Section IV, we assume that each data access on a storage node consists of a startup time and data read/write time. Normally, the startup time is independent from request size, while data read/write time is proportional to request size. We define I/O workload on each I/O server with comprehensive consideration of two parts: the total number of data accesses and the total data size accessed. Noticing that startup time for each data access is α on one node, and the transmission time of a single unit of data is β, we use K to represent the total number of data accesses and S to represent the total requested size on an I/O server. Thus, I/O workload on one I/O server can be represented as follows.

$$L_{IO} = K \cdot \alpha + S \cdot \beta \tag{5}$$

If an I/O server serves lots of small data accesses, the first part '$K \cdot \alpha$' is very large, and thus the I/O workload is very high. On the other hand, if the requested size is very large, the second part '$S \cdot \beta$' is very large, thus the I/O workload on that I/O server is also very high. Therefore, both large numbers of data accesses and large amounts of requested data can result in heavy I/O workload. The two observations are consistent with real application scenarios.

Assume the offset and size of an I/O request are f and r. It is not difficult to calculate the beginning I/O server and the ending I/O server. From the beginning to the ending, the number of data access K on each I/O server increases by one, and the data size S on each I/O server increases by the involved data size. Here the involved data size should consider fragment of the boundaries. Figure 3 demonstrates an example of how to measure K and S on all I/O servers for a request. Generally, assume the serial numbers of I/O servers are from 0 to $n-1$ (n is the total number of I/O servers), and the beginning offset of the segment is F, so the serial number of the beginning I/O server is

$$\lfloor \frac{f-F}{s} \rfloor \% n,$$

and the ending I/O server is

$$\lfloor \frac{f-F+r}{s} \rfloor \% n,$$

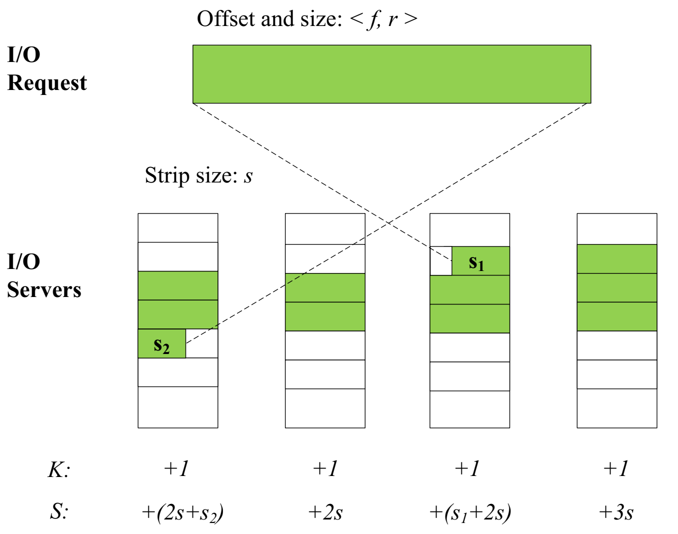


Fig. 3. A sketch of how to calculate workload for an I/O request

The size of the beginning fragment is

$$s_1 = (f-F)\%s,$$

and the size of the ending fragment is

$$s_2 = (f-F+r)\%s.$$

Therefore, if we can get all the request information from I/O traces, we can calculate the overall workload for all I/O servers in a given parallel system with different configurations of stripe sizes.

From the method of calculation we can see that, the distribution of workload depends on the data access patterns and data layout manners. Generally, data access patterns and data layout manners have an interrelated effect on I/O workload distribution. As mentioned in previous sections, in many large-scale and data-intensive applications, data accesses might be non-uniform to the file, which can introduce workload imbalance problems. Generally, workload imbalance can be measured as follows [42]:

$$\sigma = \frac{L_{max}}{L_{avg}} - 1 \tag{6}$$

where L_{max} is the maximum data access workload in all I/O servers, and L_{avg} is the average workload of these I/O servers. From the measurements, we can see that $\sigma \geq 0$. The closer σ gets to 0, the more balanced the I/O workload is. When $\sigma = 0$, the data access workload is well balanced among all I/O servers.

B. Balance I/O Workload

For a given application with specific data access patterns, stripe size is one of the key factors that influence I/O workload. For instance, in the stride I/O cases shown in Figure 1, different stripe sizes have different effects on the I/O workload over I/O servers. As a result, it is feasible to balance workload

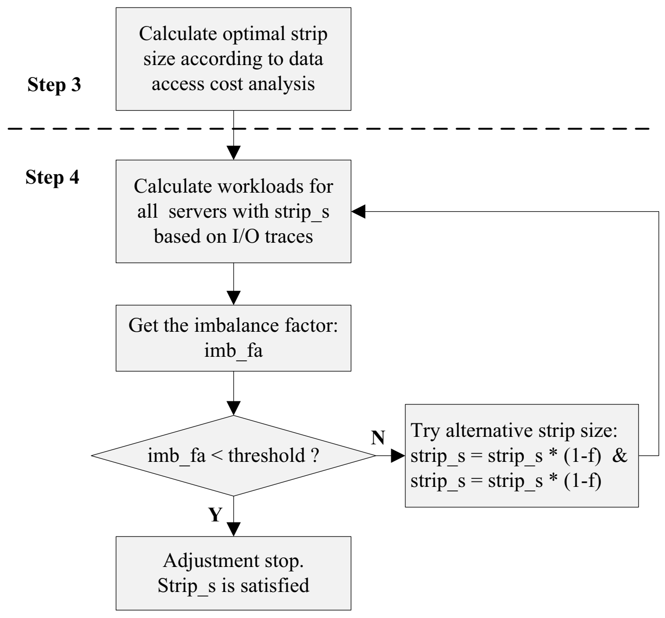

Step 3
Calculate optimal strip size according to data access cost analysis
Step 4
Calculate workloads for all servers with strip_s based on I/O traces
Get the imbalance factor: imb_fa
imb_fa < threshold ?
N
Try alternative strip size: strip_s = strip_s * (1-f) & strip_s = strip_s * (1-f)
Y
Adjustment stop. Strip_s is satisfied

Fig. 4. Balance I/O workload by adjusting stripe size

by adjusting the stripe size in an application-specific manner for large-scale and data-intensive systems.

We have discussed how to calculate the optimal stripe size of each segment based on the data access time analysis in last section. Nevertheless, the optimal stripe size is calculated from average request size, which may have lost some individual information of data accesses. It cannot guarantee I/O workload balance among multiple I/O servers. We further tune the stripe size calculated in Step (3) to achieve I/O workload balance. We calculate the workload on all I/O servers with this stripe size. If the I/O workload skew is too high, i.e., workload imbalance calculated by Formula 6 is higher than a threshold (e.g., 0.20), we use a loop-iteration method to adjust the stripe size, until the workload is well balanced. Here we set the threshold as 0.20 based on experience and rule of thumb.

Figure 4 shows how to balance workload by adjusting stripe size. We introduce an adjustment factor f $(0 < f < 1)$ in the algorithm. If workload is not balanced, the algorithm will try two alternative stripe sizes and then recalculate the workload. The adjustment of stripe size is an iterative process, and it stops when workload imbalance is less than the threshold. In order to avoid an endless loop, we set a maximum number of iterations of the adjustment. If the I/O workload is still unbalanced after all iterations, it will adopt the stripe size calculated from Step (3).

VI. Experimental Evaluation

A. Experimental Setup

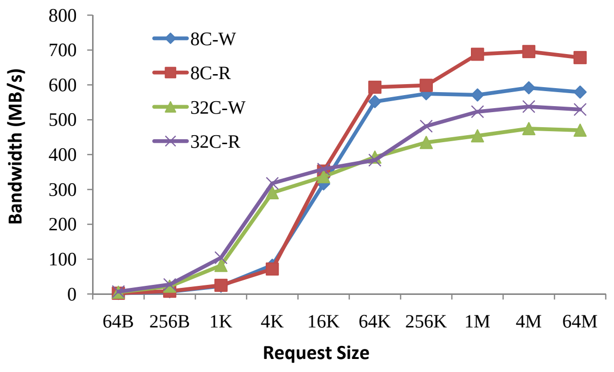
The experiments were conducted on a 65-node SUN Fire Linux cluster, in which there are 64 computing nodes and one head node. Each computing node has two Quad-Core AMD Opteron(tm) processors, 8GB memory and a 250GB 7200RPM SATA-II disk. All nodes are equipped with Gigabit Ethernet interconnection, and 17 nodes are equipped with additional 4X InfiniBand interconnection. The operating system is Ubuntu 4.3.3-5, linux kernel 2.6.28.10. The parallel file system is PVFS2 version 2.8.1. With both Ethernet and InfiniBand networks, 8 nodes are configured as I/O servers, and the other nodes are configured as I/O clients.

We evaluated the proposed layout optimization with two popular I/O benchmark tools, IOR and IOzone. IOR consists of a variety of different data access patterns, and it leverages the scalability of MPI to easily and accurately calculate the aggregate bandwidth. It is more suitable for performing I/O throughput experiments with multiple clients. IOzone is a filesystem benchmark tool for broad file system analysis, which tests I/O performance with a bunch of file operations, including ‘write’, ‘read’, ‘re-write’, ‘re-read’, etc. It is more suitable for performing single client I/O throughput experiments.

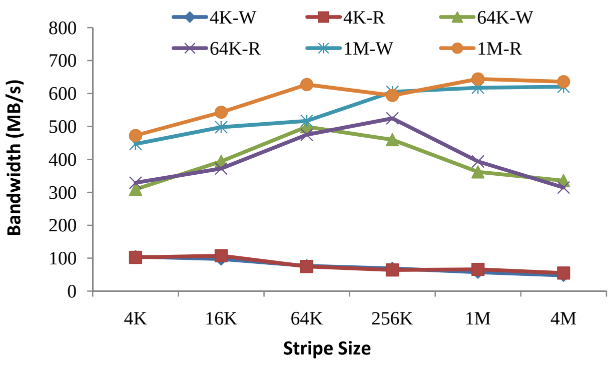
B. Results Analysis

First we verified how the I/O performance was affected by stripe size and request size in a PVFS2 system. Figure 5 demonstrates the I/O performance with different request sizes and stripe sizes in both Ethernet and InfiniBand environments. The results were collected from the IOR benchmark. In Subfigure (a) and (c), the stripe size was 4 KB. Label ‘8C-W’ means writing performance with 8 I/O clients, and ‘8C-R’ means reading performance. Other labels have similar meanings. From the results we can observe that the I/O bandwidth first rises as the request size increases, and when the request size exceeds a certain value, the I/O bandwidth becomes sustained. In Subfigure (b) and (d), the I/O client numbers are 32 and 16 respectively. ‘4K-W’ means writing performance of 4 KB request size, and ‘4K-R’ means reading performance. Other labels are similarly defined. From Subfigure (b) and (d), we can observe that stripe size and request size have an interrelated effect to the I/O performance. For example, in Subfigure(d), 64 KB stripe size can obtain the highest bandwidth when the request size is 64 KB; while 1 MB stripe size can achieve the highest performance when the request size is 1 MB or larger. This set of figures illustrate that stripe size and request size are two key related factors that affect I/O performance in parallel file systems.

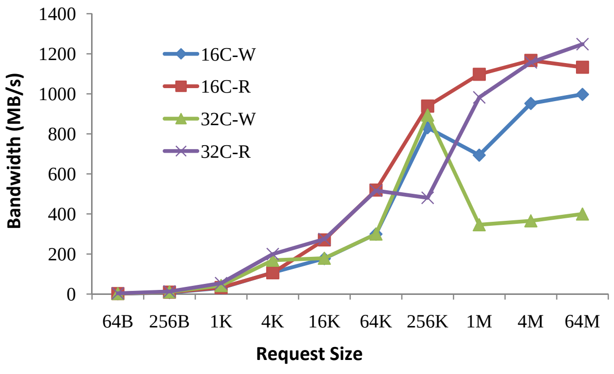
We also conducted experiments to compare the I/O performance of the proposed data layout strategy with current strategies with fixed stripe size. Figures 6 and 7 show the average write/read I/O bandwidth of different layout strategies with mixed workloads of the IOR benchmark. In this set of experiments, the data file was divided into four segments. The segment sizes were 128 MB, 1024 MB, 1024 MB and 4096 MB, respectively. We simulated the scenario of all layout manners by dividing the original file into 4 subfiles in PVFS2, one subfile for each segment and configured with different stripe sizes according to the layout strategies. For each layout strategy, we ran four instances of the IOR benchmark sequentially, each instance accessing one segment. We configured different parameters for the four instances, so that different



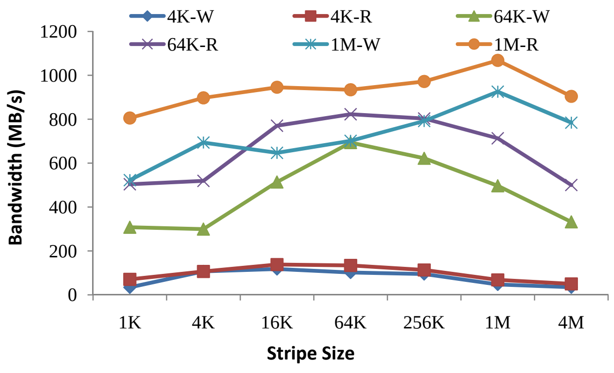
(a) Bandwidth of different request sizes with Ethernet



(b) Bandwidth of different stripe sizes with Ethernet



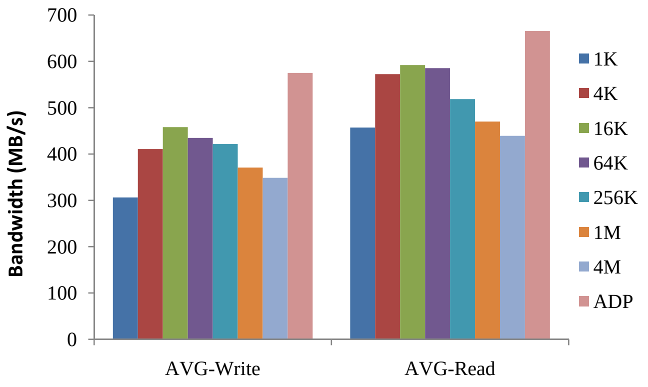
(c) Bandwidth of different request sizes with InfiniBand



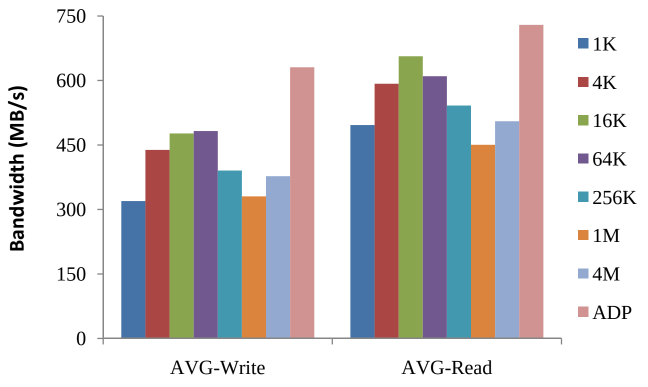
(d) Bandwidth of different stripe sizes with InfiniBand

Fig. 5. I/O performance of IOR benchmark for different request sizes and stripe sizes with InfiniBand and Ethernet interconnects

segments have different data access patterns. Each instance includes 16 I/O client processes in all the set of experiments. In Figures 6 and 7, the label 'ADP' refers to our proposed layout manner with adaptive stripe size for individual file segments, '4K' refers to that the stripe sizes were 4 KB for all segments, and other labels are similarly defined.



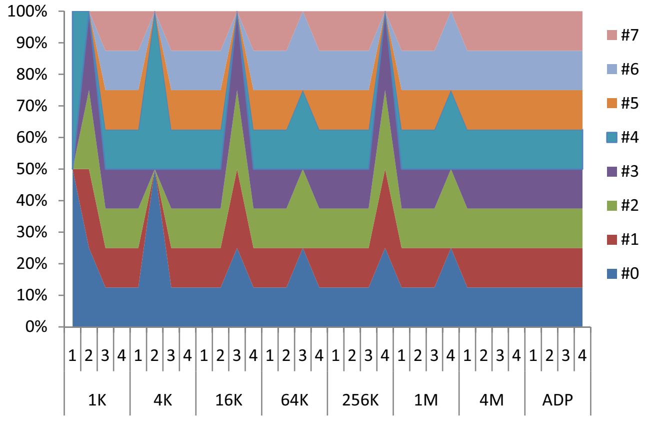
(a) Average bandwidth in Ethernet environment



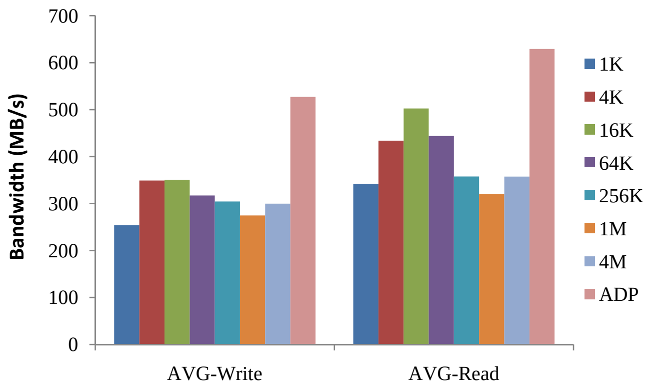
(b) Average bandwidth in InfiniBand environment

Fig. 6. Average bandwidth for sequential I/O workloads, calculated as the overall read/write data size divided by the overall running time. The labels on the right are different stripe sizes: various fixed stripe sizes, and ADP is layout proposed in this paper.

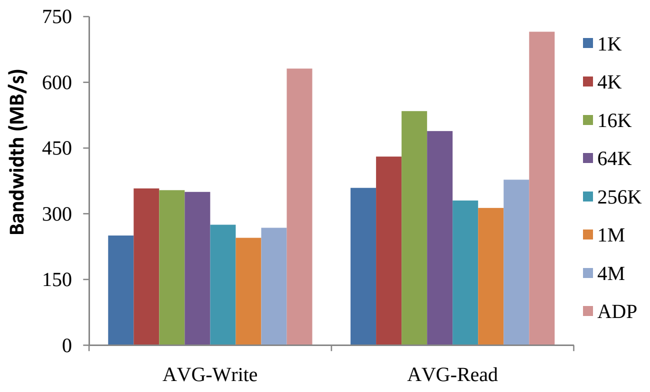
In Figure 6, as we tested the performance of sequential data access, the I/O workload was well balanced among all I/O servers. The request sizes on these segments were 1 KB, 4 KB, 64 KB and 1 MB, respectively. With the proposed segment-level layout strategy, the optimal stripe sizes of different segments were 4 KB, 16 KB, 64 KB and 1 MB, respectively. From the results, we observe that with the proposed layout scheme, the writing performance can achieve about 25% to 101% improvement, and the read performance can achieve around 9% to 71% improvement compared with other layout strategies with fixed stripe size for the whole file. We can also see that the improvement of reading is less than writing, however, the read bandwidth is higher than writing. That is because the local Linux file system on each file server uses read ahead effectively. These results show that the proposed layout scheme can achieve a significant improvement in I/O bandwidth for balanced I/O workloads.



(a) Workload distribution among I/O servers



(b) Average bandwidth in Ethernet environment



(c) Average bandwidth in InfiniBand environment

Fig. 7. Average bandwidth of strided I/O workloads. Strided I/O can introduce workload imbalance among I/O servers. Sub-figure (a) shows the workload distribution on the 8 I/O servers ('#0' $\sim$ '#7') for different layout manners. The 'x' axis of (a) represents the 4 file segments with different stripe sizes. Sub-figures (b) and (c) show the average I/O bandwidth in Ethernet and InfiniBand environments; the labels on the right are different stripe sizes: various fixed stripe sizes, and ADP is layout proposed in this paper.

In Figure 7, we tested the performance of strided data access. The request size and stride size pairs were <1KB,3KB>, <4KB, 12KB>, <64KB, 64KB> and <1MB, 1MB> for different segments, respectively. As discussed in Section V, strided I/O can introduce workload imbalance problem when adopting different stripe sizes. Figure 7(a) shows the distribution of the I/O workload for each segment in all I/O servers for different layout strategies. We see that the workload was unbalanced among the I/O servers when the stripe size was fixed for the whole file (except in the 4 MB case). The proposed layout takes workload balance into consideration, and the stripe sizes for the 4 segments were 4 KB, 16 KB, 128 KB, and 2 MB, respectively. This resulted in a balanced data access workload for all segments. Subfigures (b) and (c) demonstrate the average bandwidth in Ethernet and InfiniBand environments, respectively. We observe that with the proposed segment-level layout scheme, the writing performance can achieve about 50% to 163% improvement, and the reading performance can achieve around 26% to 132% improvement compared with other layout strategies with fixed stripe sized for the whole file. We also notice that if the fixed stripe size is 4 MB in this set of experiments, the workload is also balanced between I/O server, but the I/O performance is not good enough. That is because the stripe size is too large to benefit all kinds of data accesses. Therefore, the proposed layout with I/O workload balance awareness can achieve an even greater improvement of I/O performance for non-uniform data access.

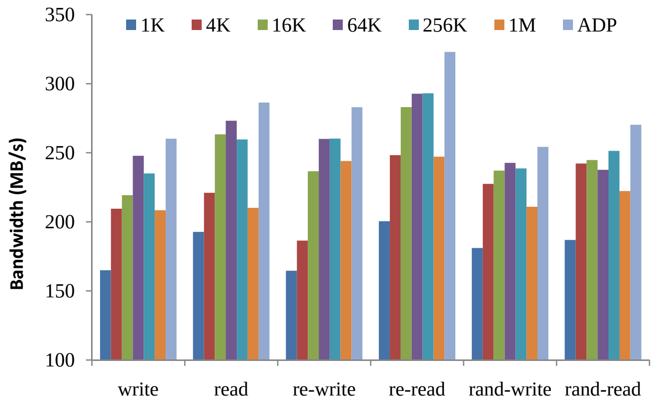


Fig. 8. Average I/O bandwidth for the IOzone benchmark. We ran 1 IOzone client with 8 I/O servers. The file was divided into 3 segments of size 64 MB, 256 MB, and 4GB. Request sizes on these segments were 4 KB, 256 KB, and 4 MB, respectively.

Figure 8 demonstrates the average bandwidth with IOzone benchmark. The results were tested with PVFS2 POSIX interface. The IOzone benchmark was run on only one client node, and we tested the I/O bandwidth only in InfiniBand environment so that the network bandwidth is not the bottleneck of data access. The testing file was divided into 3 segments of size 64 MB, 256 MB, and 4 GB. The request sizes on these segments were 4 KB, 256 KB, and 4 MB, respectively. We measured the I/O bandwidth with 'write', 'read', 're-write', 're-read', 'rand-write', and 'rand-read' operations, in which 'rand-write' and 'rand-read' mean random write and random read. The stripe sizes of the segmented layout scheme for the three segments were 16 KB, 64 KB and 256 KB, respectively. We compared the I/O bandwidth for different layout strategies. From the results, we can observe that the proposed layout scheme obtained the highest I/O bandwidth, and the performance improvement is around 6%∼57% compared with other layout strategies with fixed stripe sized for the whole file.

From the results of sequential and strided I/O of IOR experiments, we see that the proposed layout scheme can achieve significant improvement in I/O performance for both cases. Furthermore, the new layout scheme with workload balance awareness can get more potential benefit from the unbalanced workload. By comparing the experimental results of IOR and IOzone benchmark, we observe that the performance improvement in IOR experiments is higher than that in IOzone experiments. The reason is that with multiple clients in the IOR benchmark, the I/O workload is more heavy on I/O servers, and there is more contention on storage devices. For this reason, we can tentatively conclude that the proposed layout scheme can achieve more performance improvement in heavy I/O workloads.

VII. Conclusions and Future Work

Data access patterns and layout strategies have an interrelated effect on the I/O performance of parallel file systems. Existing strategies that use a fixed stripe size for the entire scope of a file may not obtain the best I/O performance, especially for applications with varying data access patterns across a large file. In large-scale parallel I/O systems, a single file could reach several terabytes, and data access patterns to the file could vary at different parts of the file. Moreover, non-uniform data access could introduce workload imbalance problems among I/O servers. In this paper, we proposed a novel segment-level data layout strategy for large-scale and data-intensive applications. This new strategy can provide a fine-grained segment-level data layout optimization with data access workload balance awareness, which is more suitable for applications with non-uniform data access patterns.

We described the five-step approach of the proposed data layout optimization, including how to divide the file into segments, how to calculate the optimal stripe size for each segment, and how to balance the I/O workload among I/O servers for each segment. The proposed data layout scheme adopts different stripe sizes for different segments, and refines layout optimization at the segment level. It achieves a better integration of data access characteristics of applications and data organization in parallel file systems. The experimental results demonstrate that, the proposed segmented layout strategy with workload balance awareness improves the performance up to 163% for writing and 132% for reading on the widely-used IOR and IOzone benchmarks.

In the future, we plan to refine the data access cost model to estimate the optimal stripe size of each segment more precisely. We also plan to define some specific segment-level layout templates for common applications, which could guide users to choose the optimal layout for them.

Acknowledgment

The authors are thankful to Hui Jin of Illinois Institute of Technology and Dr. Robert Ross of Argonne National Laboratory for their constructive and thoughtful suggestions toward this work. The authors are also grateful to anonymous reviewers for their valuable comments and suggestions. This research was supported in part by National Science Foundation under NSF grant CCF-0621435 and CCF-0937877, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] "High-performance Storage Architecture and Scalable Cluster File System," Lustre File System White Paper, December 2007.

[2] F. Schmuck and R. Haskin, "GPFS: A Shared-disk File System for Large Computing Clusters," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2002, p. 19.

[3] G. Gibson, B. Welch, G. Goodson, and P. Corbett, "Parallel NFS Requirements and Design Considerations," Internet Draft, October 2004. [Online]. Available: http://bgp.potaroo.net/ietf/idref/draft-gibson-pnfs-reqs/

[4] I. F. Haddad, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux Journal*, p. 5, 2000.

[5] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas Activescale Storage Cluster: Delivering Scalable High Bandwidth Storage," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 53.

[6] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–17.

[7] X.-H. Sun, Y. Chen, and Y. Yin, "Data Layout Optimization for Petascale File Systems," in *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*. New York, NY, USA: ACM, 2009, pp. 11–15.

[8] H. Song, X.-H. Sun, H. Jin, and Y. Chen, "Trace-based Adaptive Data Layout Optimization for Parallel File Systems (poster presentation)," in *PDSW '10: Proceedings of the 5th Annual Workshop on Petascale Data Storage*, 2010.

[9] H. Song, X.-H. Sun, Y. Yin, and Y. Chen, "A Cost-based Application-specific Data Layout Scheme for Parallel File Systems," in *HPDC'11: Proceedings of the 20th International ACM Symposium on High Performance Distributed Computing*, 2011.

[10] D. Kotz and N. Nieuwejaar, "File-system Workload on a Scientific Multiprocessor," *IEEE Parallel Distrib. Technol.*, vol. 3, pp. 51–60, March 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=613774.613889

[11] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur, "Passion: Parallel and Scalable Software for Input-output," Tech. Rep., 1994.

[12] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO," in *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*. Washington, DC, USA: IEEE Computer Society, 1999, p. 182.

[13] R. Bordawekar, J. M. del Rosario, and A. Choudhary, "Design and Evaluation of Primitives for Parallel I/O," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1993, pp. 452–461.

[14] 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.

[15] J. M. May, *Parallel I/O for High Performance Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[16] Y. Chen, X.-H. Sun, R. Thakur, H. Song, and H. Jin, "Improving Parallel I/O Performance with Data Layout Awareness," in *Cluster '10: Proceedings of the IEEE International Conference on Cluster Computing 2010*. Washington, DC, USA: IEEE Computer Society, 2010.

[17] A. Ching, A. Choudhary, K. Coloma, W.-k. Liao, R. Ross, and W. Gropp, "Noncontiguous I/O Accesses through MPI-IO," in *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 104.

[18] A. Ching, A. Choudhary, W.-k Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File Systems," *Cluster Computing, IEEE International Conference on*, vol. 0, p. 326, 2003.

[19] B. Nitzberg and V. Lo, "Collective Buffering: Improving Parallel I/O Performance," in *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1997, p. 148.

[20] W.-k Liao, A. Ching, K. Coloma, A. Choudhary, and Lee Ward, "An Implementation and Evaluation of Client-side File Caching for MPI-IO," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 49, 2007.

[21] X. Ma, M. Winslett, J. Lee, and S. Yu, "Faster Collective Output through Active Buffering," in *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 151.

[22] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," in *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1997, pp. 21–21.

[23] N. Tran and D. A. Reed, "Automatic Arima Time Series Modeling for Adaptive I/O Prefetching," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 4, pp. 362–377, 2004.

[24] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp, "Hiding I/O Latency with Pre-execution Prefetching for Parallel Applications," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–10.

[25] 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 *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[26] D. Kotz, "Disk-directed I/O for MIMD Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 1, pp. 41–74, 1997.

[27] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-directed Collective I/O in Panda," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1995, p. 57.

[28] W. B. Ligon III and R. B. Ross, "Implementation and Performance of a Parallel File System for High Performance Distributed Applications," in *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 1996, p. 471.

[29] R. B. Ross and W. B. Ligon III, "Server-side Scheduling in Cluster Parallel I/O Systems," *Calculateurs Parallles Journal Special Issue on Parallel I/O for Cluster Computing*, 2001.

[30] S. Rubin, R. Bodík, and T. Chilimbi, "An Efficient Profile-analysis Framework for Data-layout Optimizations," *SIGPLAN Not.*, vol. 37, no. 1, pp. 140–153, 2002.

[31] Y. Wang and D. Kaeli, "Profile-guided I/O Partitioning," in *Proceedings of the 17th annual international conference on Supercomputing*, ser. ICS '03. New York, NY, USA: ACM, 2003, pp. 252–260. [Online]. Available: http://doi.acm.org/10.1145/782814.782850

[32] W. W. Hsu, A. J. Smith, and H. C. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 424–473, 2005.

[33] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reorganization for Self-optimizing Storage Systems," in *In FAST '09: Proccedings of the 7th conference on File and storage technologies*, San Fancisco, CA, USA, 2009, pp. 183–196.

[34] H. Huang, W. Hung, and K. G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," in *Proceedings of the 20th ACM symposium on Operating systems principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 263–276. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095836

[35] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *Trans. Storage*, vol. 6, pp. 13:1–13:26, September 2010. [Online]. Available: http://doi.acm.org/10.1145/1837915.1837921

[36] X. Zhang and S. Jiang, "InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication," in *Proceedings of the 24th International Conference on Supercomputing*, 2010, pp. 223–232.

[37] A. Konwinski, J. Bent, J. Nunez, and M. Quist, "Towards an I/O Tracing Framework Taxonomy," in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07*, ser. PDSW '07. New York, NY, USA: ACM, 2007, pp. 56–62. [Online]. Available: http://doi.acm.org/10.1145/1374596.1374610

[38] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Scalable I/O Tracing and Analysis," in *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*. New York, NY, USA: ACM, 2009, pp. 26–31.

[39] P. Lu and K. Shen, "Multi-layer Event Trace Analysis for Parallel I/O Performance Tuning," in *Proceedings of the 2007 International Conference on Parallel Processing*, p. 12, 2007.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[41] "HDFS Architecture Guide," HDFS 0.21 Documentation. [Online]. Available: http://hadoop.apache.org/hdfs/docs/r0.21.0/hdfs_design.html

[42] E. M. Gengbin Zheng, Abhinav Bhatele and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IHHPCA)*, 2010.